ONTOLOGY CREATION FOR WIRELESS CAPSULE ENDOSCOPY VIDEOS

Abstract

In this paper we study multimedia ontology for Wireless Capsule Endoscopy (WCE) videos by enhancing its existing data structure. The 'wireless capsule' is a tiny disposable video camera that transmits 2 ~ 3 frames per second for a period of 8 ~ 11 hours. There are open problems in WCE, such as bleeding detection, as it is hard to identify accurately, using low-level features, i.e., color values. In addition, the physicians have to examine the videos continuously for two hours or more, which becomes restrictive. There have been research attempts to reduce this review time. However, they suffer from low accuracy and sensitivity, and do not process WCE videos with an efficient information data structure. To address this problem, we propose a new data structure named 'multimedia ontology for WCE videos' formed by combining medical and multimedia domain knowledge. Ontology represents a structure to describe the concepts and relationships in a specific domain with relevant data and its terminology. we define two types of ontology, i.e., generic and specific ontology. Generic ontology represents the broad concepts in WCE videos, such as medical terms, anatomic information, video format, etc., while specific ontology is a data-driven one including color, location, and region of images. The process of creating multimedia ontology consists of three steps: (1)collection of raw data from WCE videos, such as video data format, feature values, meta-data information and anomalies, (2) classification of the raw data into concepts including generic and specific ontology, and (3) identification of relationship between two concepts such as 'Is-A', 'Part-Of', and 'Has-A'. This WCE Ontology structure can be used to better address the open problems by providing 'relevant area focus' from the formed structure and can also be extended to other problems like detection of lesions and polyps.

Keywords: *Multimedia ontology, Generic ontology, Specific ontology, Wireless capsule Endoscopy (WCE)*

Software's Used: *Java Net Beans IDE 6.5, Microsoft Visio 2007, Protégé 3.4, Inference engines: Algernon and Jess 7.1, windows vista*

1. Multimedia Ontology

Ontology is a model of information in a given data domain that can be used for the purpose of enterprise integration, database design, information retrieval, and information exchange via World Wide Web [12, 13]. Although the term, 'ontology' originates in the philosophical study of ontology, the usage of the term is widespread in not only artificial intelligence and knowledge representation communities, but in most of information technology areas. In particular, ontology has become a common semantic, in order to categorize large taxonomies on the web and integrate the data. The reasons for using ontology in information retrieval are mentioned in [12]: (1) Domain information can be sharable between human and machine, (2) Domain knowledge can be reused, (3) Domain knowledge can be analyzed and be more explicit, and (4) Domain knowledge can be separated and merged with other [14].

However, since a typical ontology is based on *terms* for concepts, it is not helpful to use it for multimedia data. Generally, multimedia data have a lot of contents, but the data format is not well-structured. In order

to capture the characteristics of multimedia data structure and contents, we propose a multimedia ontology as follows:

> ***Definition 1***. *Given multimedia data, **M**, a multimedia ontology O_M is a three-tuple $O_M = \{C, R, \nu\}$, where*
>
> - ***C** is a set of concepts describing **M**,*
> - ***R** is a set of relationships between two concepts, and*
> - *ν is a set of functions generating attribute values of **C**.*

In Definition 1, a concept ***C*** can be represented by slot. *Slot* is an attribute or properties of a concept. For example, a concept, 'Video' may have a name, identification, format, and size as slots. Also, a concept can have data, called *Instance*, which is a concrete occurrence of information about a domain. For example, a sample WCE video might be an instance for 'Video' concept.

The main difference between typical ontology and multimedia ontology is a set of functions (ν) generating attribute values of concepts in multimedia ontology. Using the attribute functions, various feature values are extracted from the multimedia, such as color, shape and region in video data. Therefore, it is required that a part of multimedia ontology can be constructed by using a data-driven approach. In other words, some concepts in multimedia ontology are generated mainly from data while the other concepts are created without data processing.

Based on the multimedia ontology defined in Definition 1, now we can have multimedia ontology for WCE video when ***M*** is a WCE video, i.e., O_{WCE}. In the multimedia ontology for WCE videos, O_{WCE}, we define two types of ontology, i.e., generic and specific ontology. Generic ontology represents the broad concepts in WCE videos, such as medical terms, anatomic information, video format, etc., while specific ontology is data-driven one including color, location, and region of images.

2. Creating Multimedia Ontology for WCE videos

In this section, we are going to construct multimedia ontology for WCE videos, i.e., O_{WCE}. The proposed multimedia ontology can be created by following the three steps: (1) collection of raw data, (2) selection of concepts, and (3) identification of relationships. Figure 2 shows the overall procedure of creating multimedia ontology for WCE videos.

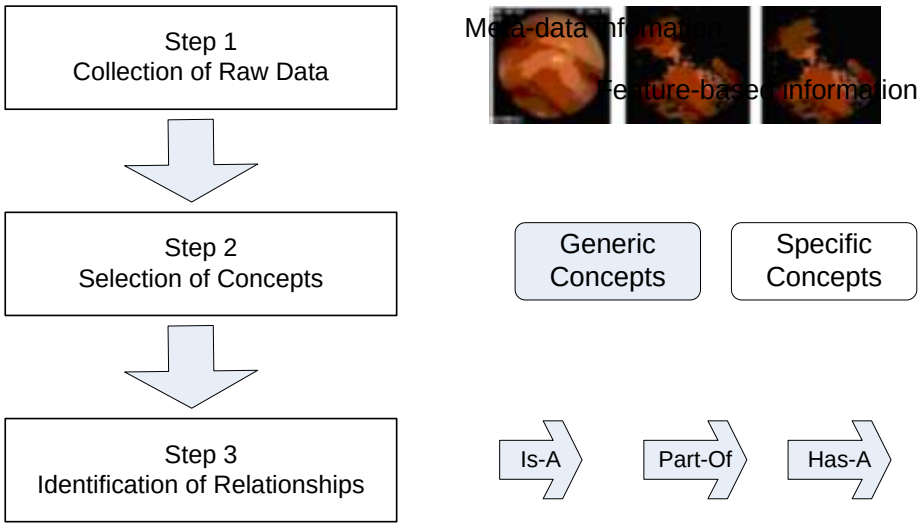


Figure 1 Overview of creating multimedia ontology for WCE videos (O_{WCE})

2.1. Collection of raw data

As mentioned in Section 2, the main difference between multimedia ontology and traditional ontology is that the former manages the extracted information from the multimedia data while the later is based on terms. Therefore, the proposed procedure of creating multimedia ontology for WCE videos starts over the collection of raw data in WCE videos. Since some conceptual information managed in the multimedia ontology is data-driven, we need to collect all information about WCE. The raw data can be obtained by the following ways:

- Meta-data information:
  Patient (ID, Age, Sex, Diseases),
  Operation (Hospital, Doctor, Date, Time, Year),
  Capsule (Manufacturer, Video ID),
  Anatomy (Digestive Organ, Diseases, Features),
  Video (Format, Length, Resolution, Frame Rate).

- Feature based information:
  Events (Event ID, Start Frame, End Frame, Anatomy),
  Color (Diseases, Organ, Feature Values, Threshold Value),
  Shape (Diseases, Organ, Feature Values, Threshold Value),
  Region (Diseases, Organ, Feature Values, Threshold Value)

Table 1 shows the details about each raw data and the method that is used for extracting the data from WCE videos.

Table 1. Details of raw data extracted from WCE videos

	Raw data	Attributes	Method
Meta-data information	Patient	ID, Age, Sex, Disease	Expert
	Operation	Hospital, Doctor, Date, Time, Year	Expert
	Capsule	Manufacturer, Video ID	Expert
	Anatomy	Organ, Diseases	Expert
	Video	Format, Length, Resolution, Frame rate	Properties of Video
Feature-based information	Event	ID, Start frame, End frame, anatomy	Event boundary detection
	Color	Feature values, threshold value	Color histogram
	Region	Feature values	Region segmentation
	Shape	Feature values	Shape detection

2.2. Selection of concepts

After the raw data of WCE videos are collected, the next step is the selection of concepts to be used for the multimedia ontology. As we discussed before, multimedia ontology has two types of ontology, i.e., generic ontology and specific ontology. In this selection step, we determine the specific ontology using the collected data. Since concepts in generic ontology are common in any multimedia ontology for WCE videos, it is straightforward way to select such a generic ontology from existing one. However, selection of concepts in specific ontology is much more complex and difficult than that in generic one. In order to select the concepts in specific ontology, we need to analyze the data extracted in the previous step. For the analysis of raw data, we group the extracted raw data into a concept based on relevance. To determine the relevance of feature values, we can utilize several data mining techniques, such as clustering and decision tree. Then, we can create instances for each selected concepts that describe the WCE videos. Figure 3 shows an example of selected concept, i.e., 'Event', and its instances. 'Event' concept has several slots, such as Event ID, Start Frame, End Frame, Organ, and Abnormal. 'Organ' and 'Abnormal' are also concepts in WCE ontology.

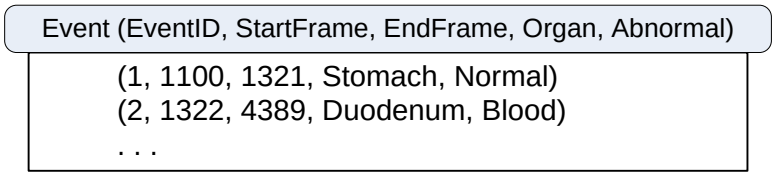


Figure 2 Example of selected concept and its instances

2.3. Identification of relationships

In Section 2, we define three types of relationships in multimedia ontology for WCE videos, i.e., (1) 'Is-A', (2) 'Part-Of', and (3) 'Has-A'. *Is-A* represents a membership relationship to define which concept is a member of its super-concept. For example, two concepts "Frame" and "Video" has *Is-A* relationship since a frame is a member of video. On the other hands, *Part-Of* relationship describes how multiple concepts form a composite concept, such as stomach is *Part-Of* digestive organ. The last relationship used in

multimedia ontology is *Has-A* that indicates the ownership between two objects, for example, "Patient" *Has-A* polyp of certain Color. Generally, *Has-A* is a dynamic relationship that can be identified by expertise based on the existing information. Figure 4 shows examples of identified relationships in WCE videos. For example, "Event" and "Abnormal" have *Has-A* relationship since each event may or may not have abnormal events that are defined as separated concepts.

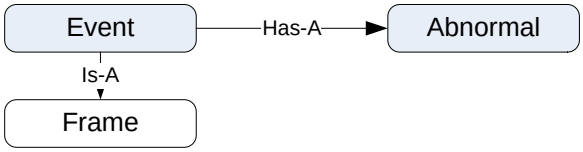


Figure 3 Example of identified relationships

3. Case Study of Multimedia Ontology for WCE videos

In order to justify the proposed multimedia ontology, we have studied several WCE videos to create the ontology as a case study. In order to build concepts of generic ontology, we employ standard terminology commonly used in endoscopic operation. For the standard, we use Minimal Standard Terminology (MST) that is a controlled vocabulary for colonoscopy and endoscopy reporting [11]. Some generic concepts from MST are given below:

- Upper GI tract organs: Esophagus, Stomach, Duodenum, Jejunum
- Lower GI tract organs: Colon, Ileum
- Blood: Red, Clot, Hematin (altered blood)
- Other contents: Food, Fluid, Foreign Body, Stent
- Lesions: Spot, Dieulafoy lesion, Angioectasia
- Protruding lesions: Enlarged folds, Papule, Polyp, Tumor, Varices, Suture granuloma
- Mucosa: Erythematous, Congested, Granular, Friable, Nodular, Hemorrhagic
- Normal: Normal
- Abnormal: Blood, Pus, Sludge, Impacted stone, Foreign body, Parasites, Stent, Drain
- Diseases: Tumor, Gastro-esophageal reflux disease, Ulcer, Gastritis, Stenosis, Gastrointestinal bleeding, Varices, Precancerous lesions, Foreign Body, Metastasis of unknown origin

Also, the following concepts are selected for the specific ontology based on the extracted raw data from WCE videos.

- Video: Format, Length, Resolution, Frame Rate
- Patient: ID, Age, Sex, Diseases
- Events: Event ID, Start Frame, End Frame, Anatomy
- Color: Diseases, Organ, Feature Values, Threshold Value
- Shape: Diseases, Organ, Feature Values, Threshold Value
- Region: Diseases, Organ, Feature Values, Threshold Value

Then, we identify relationship between (or among) concepts that are used for multimedia ontology as follows:

- A video is a WCE video.

- Endoscopy operation is a WCE video.
- Endoscopy operation has an organ, disease, and anomaly.
- Upper (Lower) organ is a part of digestive organs.
- Esophagus (Stomach, Duodenum, Jejunum) is a part of upper organ.
- Colon (Ileum) is a part of lower organ.
- Pas (Stone, Drain, Blood) is a part of abnormal
- Red (Clot, Hematin) is a part of Blood.
- Video has a patient.
- An event is a video.
- A frame is an event.
- A feature is a frame.
- Color (Region, Shape) is part of a feature.
- Abnormal has a feature.
- Organ is an event.
- Organ has a feature.

Based on the selected concepts and identified relationships from WCE videos, the multimedia ontology for WCE videos is represented with a tree-like structure as shown in Figure 5. In Figure 5, rounded rectangle with color represents generic ontology while white colored one indicates specific ontology.

The created multimedia ontology in Figure 5 can be used for various applications that require high-level approach. For example, to detect bleeding in WCE videos, we can retrieve some relevant data from the ontology to enhance the accuracy.

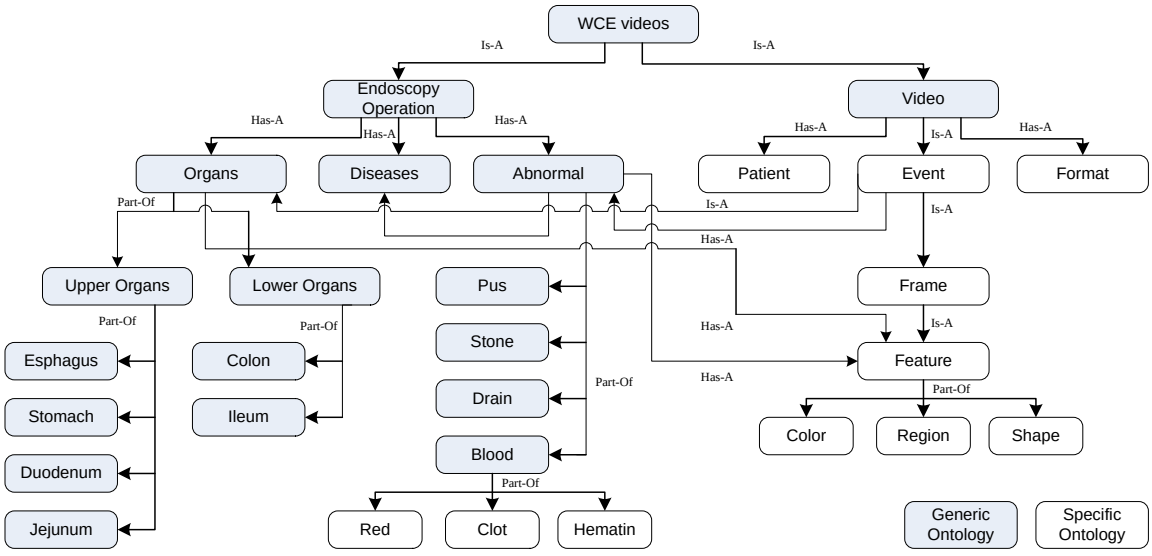


Figure 4 Multimedia ontology for WCE videos (O_{WCE})

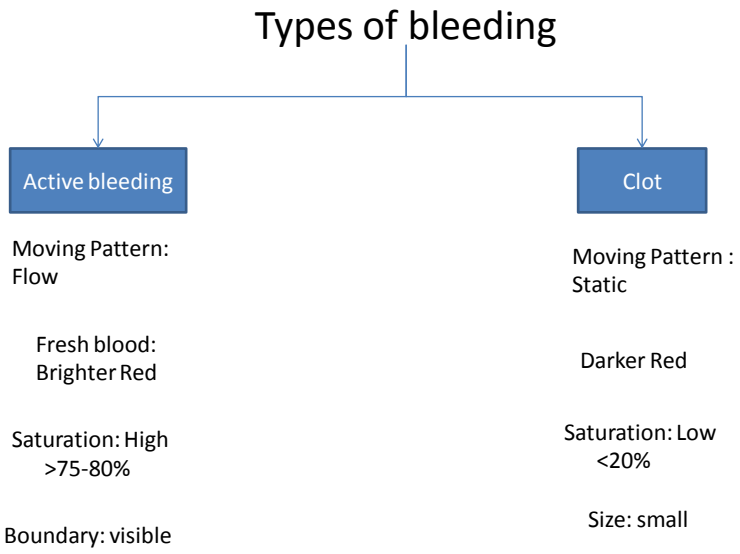
Example: Retrieval from the multimedia ontology for WCE videos (O_{WCE})

Upon review of the video from WCE, the physician accesses the ontology structure that has been created, and checks whether any abnormalities occurred or not with the help of the 'Abnormal' (Event) located in 'generic ontology' under the endoscopy operation. Abnormality is triggered if any one of the conditions is

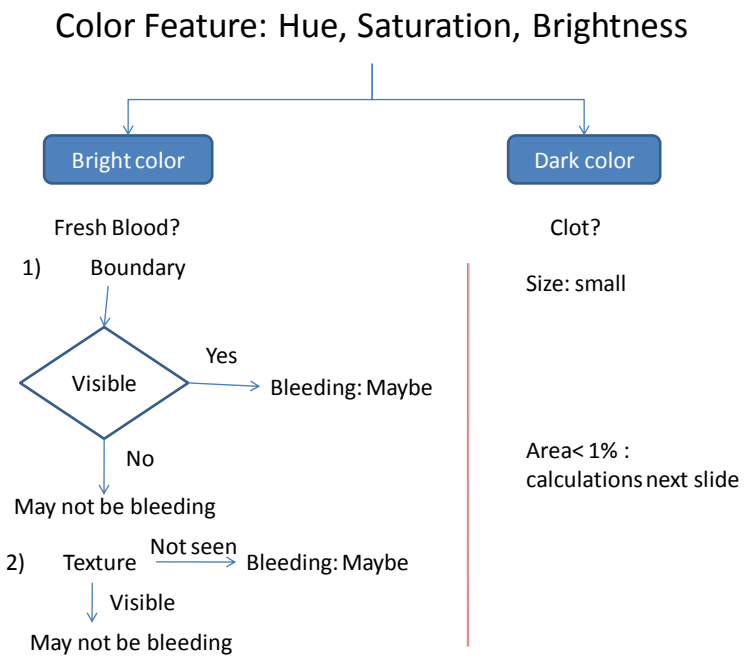
satisfied: 1) Color of the blood is Red, or 2) Clot formation or 3) 'Hematin' is found. In this case, when the color is found to be red then there exists some bleeding. To check which part of the organ is affected the physician traverses through the Organ, which is in generic side of endoscopy operation, and finds out whether it is in upper or lower part of the organ. With the help of the video images referring to the specific ontology, the specialist collects the patient information. Since the abnormality was triggered, Event has occurred. Then the specialist proceeds to check for the feature and threshold values. If it exceeds the set range, then the doctor concludes that bleeding has occurred in that particular region. Therefore, with the help of the above multimedia ontology structure we can see that information is retrieved quicker and it helps in faster diagnosis.

Types of Bleeding:

Based upon the wireless capsule endoscopy videos: Bleeding has been classified as three types:



Color Feature: Based on color values its classified as Hue, Saturation and Intensity



Area calculations:

How to calculate area and its assumptions

From the image we can get width and height:
Width~48
Height~112
48*112=5376
Whole frame=(320*320)= 602112
Four triangle cut off corners((85*85/2)*4)= 14450
Total= 587662(602112-14450)
Actual image= 0.0091(5376/587662)

Therefore **area < 1%** size is small

Algorithm steps for identifying bleeding:

Pseudo code for bleeding

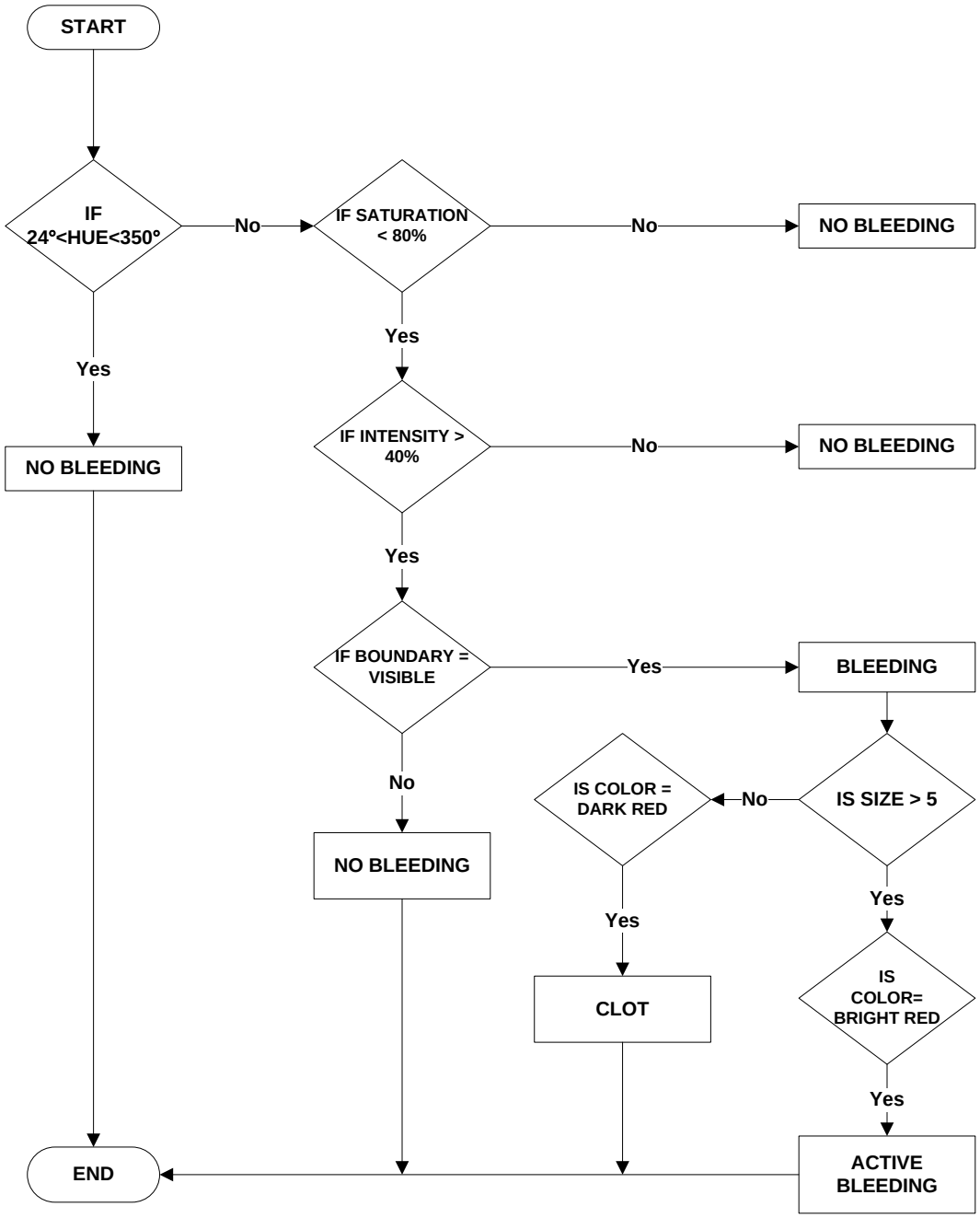
```
In a color wheel, Hue has specific ranges:
If (22°<hue<350°) // 0.96-1 in pink side & 0.038-0.055 in red side
not bleeding
Else
        IF (saturation < 80%)
        Not bleeding
                Else
                        //Could be active bleeding
                        IF (intensity > 40% )
                                IF (boundary exists) Bleeding occurred
                                Else        No bleeding
                                 ENDIF
                         ENDIF
        ENDIF
ENDIF
```

Not considered:

- Blood group
- Video light decreases: blood color may change

Flowchart:

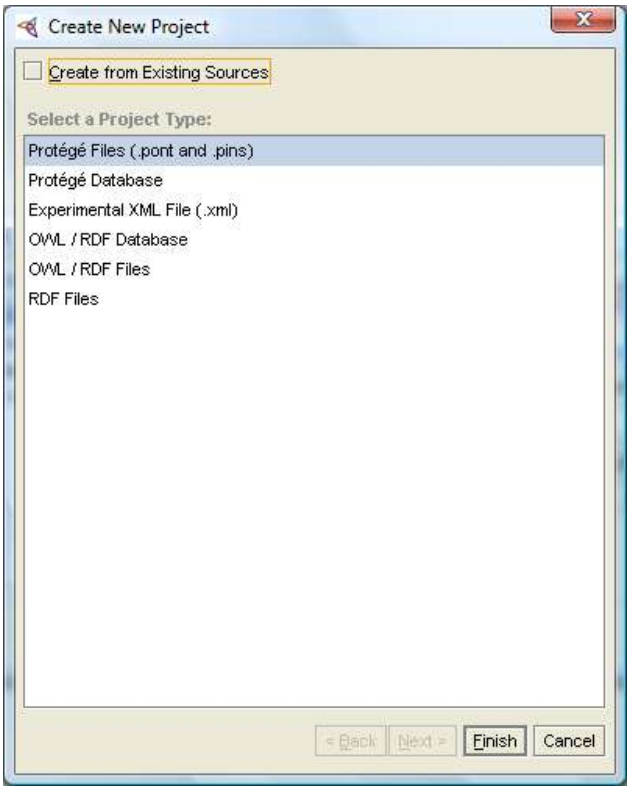
I have attached a flowchart which identifies as how to identify active bleeding or clot and it was done in Microsoft visio drawing.



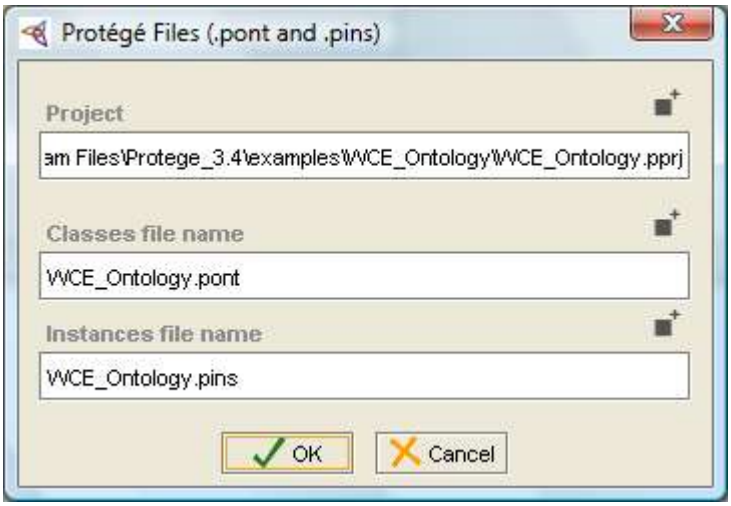
Obtain Protégé
Download from http://protege.stanford.edu/

Obtain Jess
Download from http://www.jessrules.com/
To obtain full version of Jess license its free for academic use

How to open a new project and save the project in protege:
Open--→ new project-→ default project selected -→ click finish.



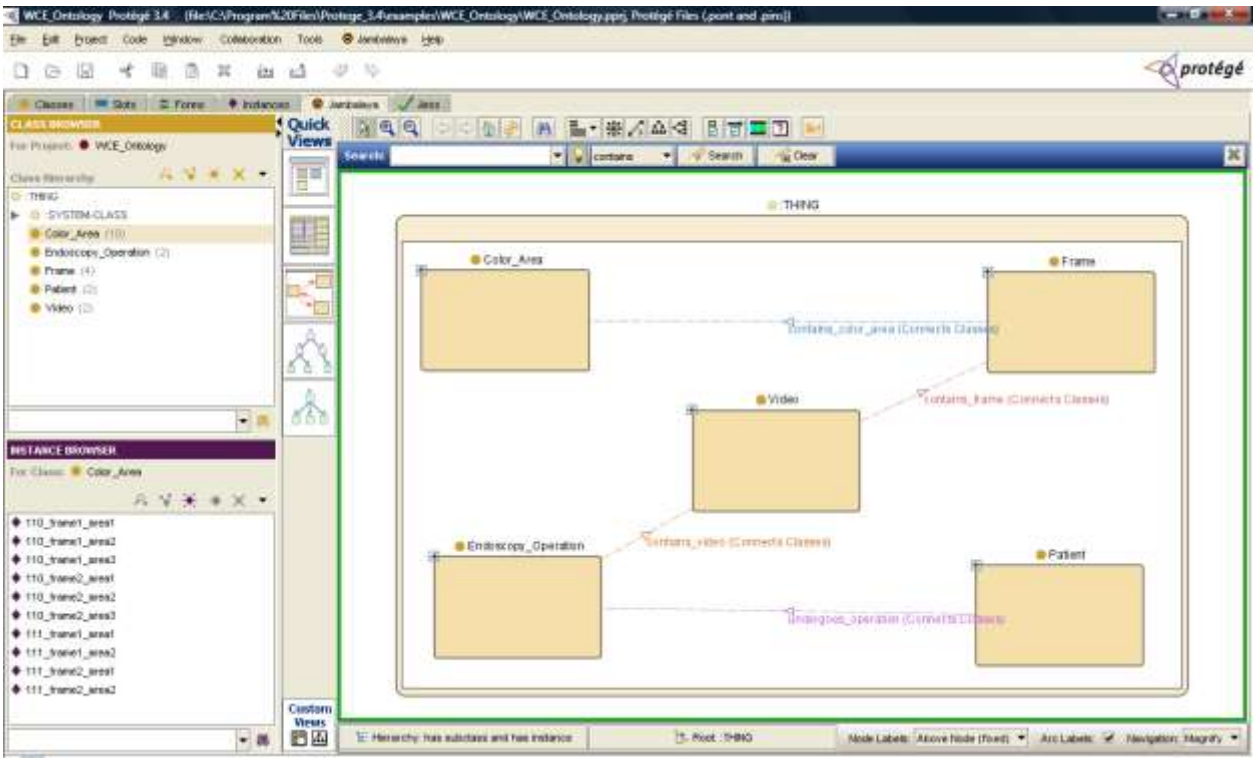
File→save project→ click the square+ icon above project to set the path and give it a name and file type as .pprj.
It will automatically create the other .pont and .pins files



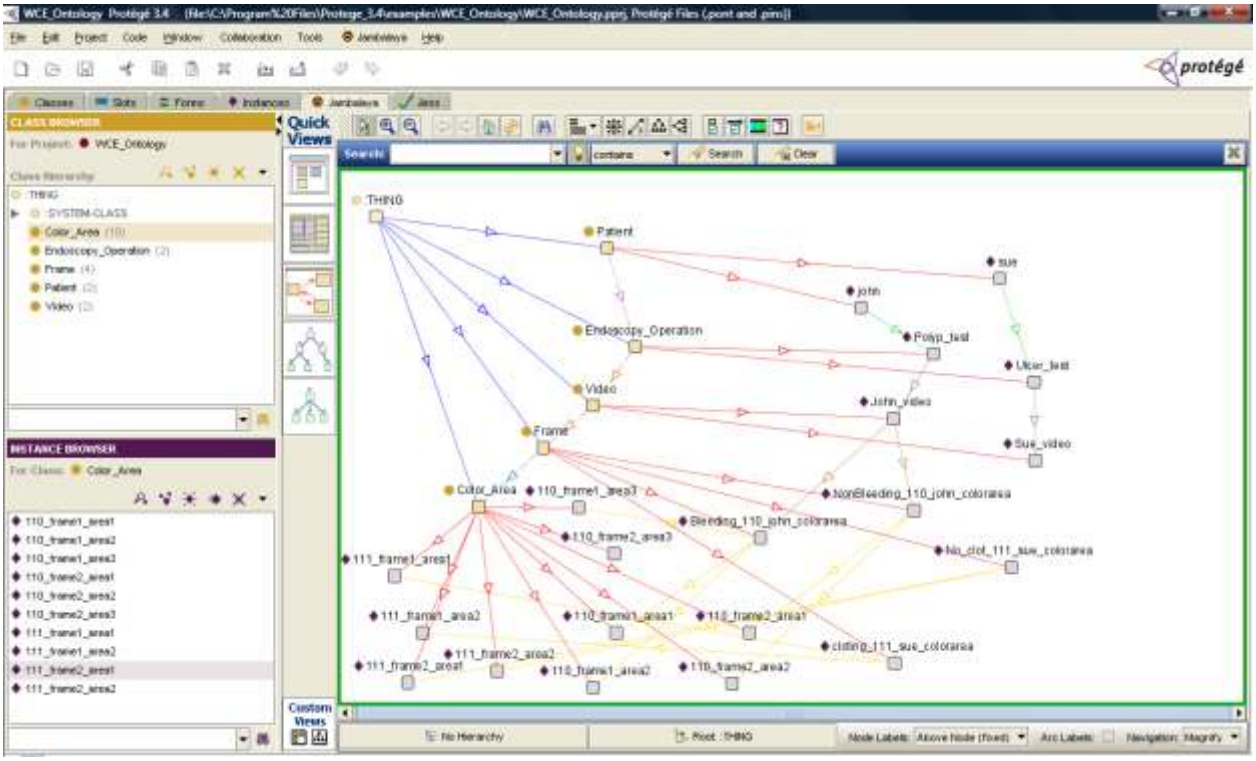
Various types of plug-in available to represent ontology:

I used jambalaya which is user friendly. Other plug-in like owl viz, are also available to represent the ontology data structure.

Structure of WCE in protégé: (jambalaya)



Tree structure:



I tried two of inference engine rules:

1) Algernon
2) Jess

Algernon - Rule-Based Programming

The Algernon rule-based inference system is now implemented in Java and interfaced with Protégé. Algernon performs forward and backward rule-based processing of frame-based knowledge bases, and efficiently stores and ret

Example;: rule based

Known problems with this release

- The :TAXONOMY command doesn't correctly return all of its variables. However, all elements in the taxonomy are correctly created.
- Output from the LOAD button shows up in the console window rather than in the Protege tab.
- Output from :PRINTLN commands shows up in the console window rather than in the Protege tab.
- :LIST type is not supported.
- Algernon does not notice KB changes made by hand via the Protege gui.
- While you can now load rules stored in the KB, you can only load one rule set at a time.
- :FORC-INSTANCE and :FORC-CLASS do not work correctly yet.

Algernon:protégé:

Input query:

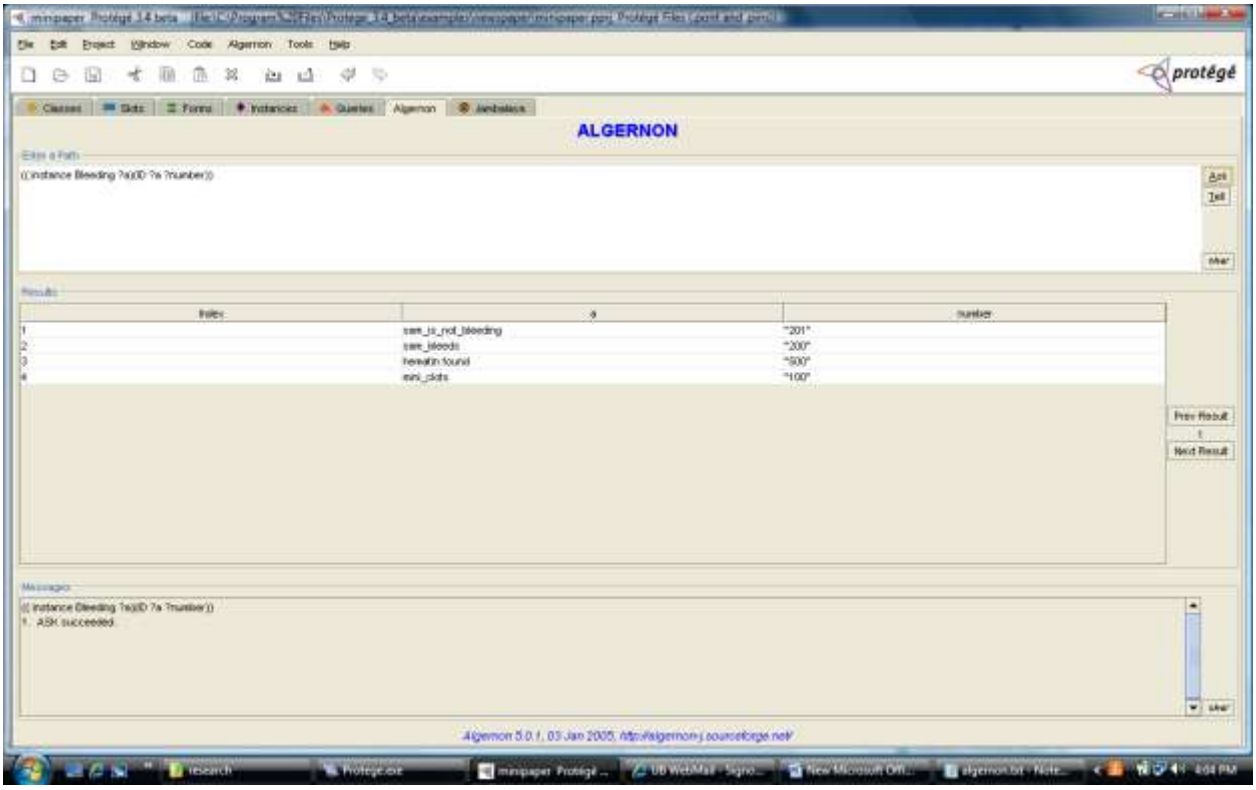
((:instance Bleeding ?a)(ID ?a ?number))

Output:

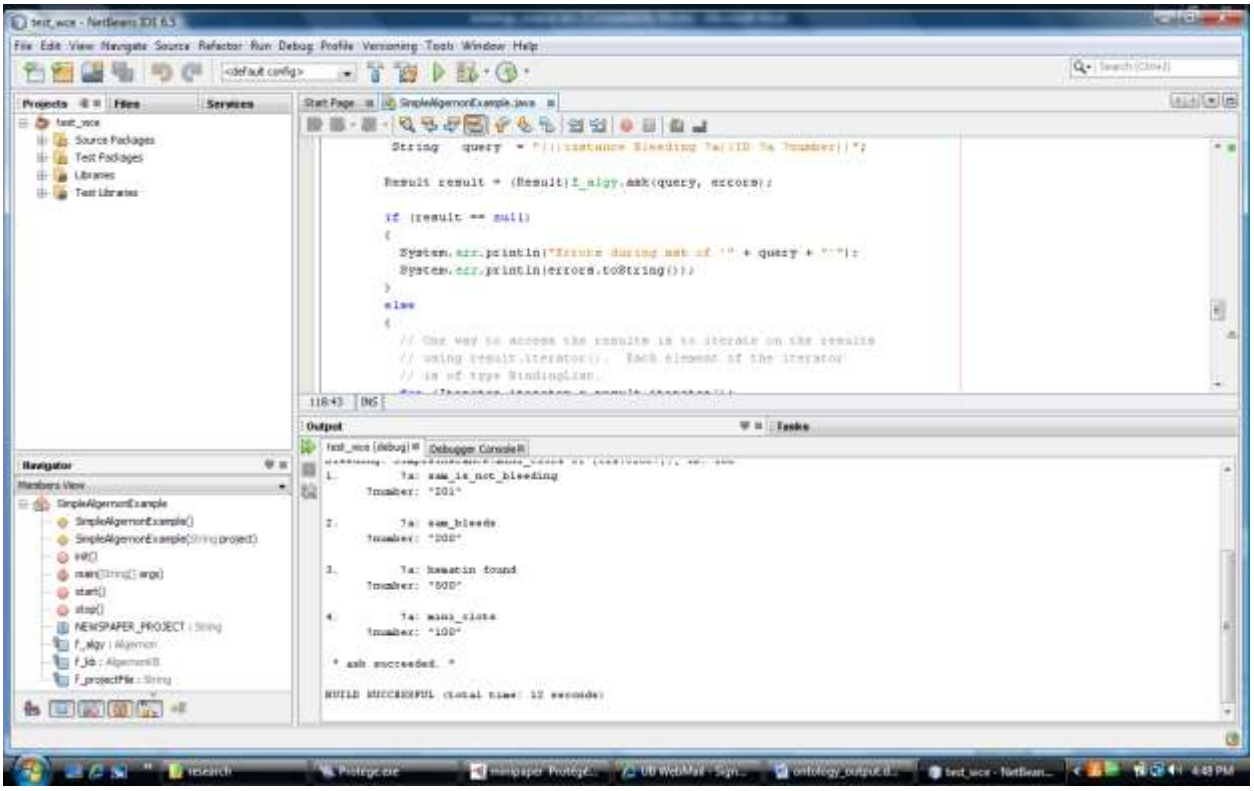
1 minipaper_Class21 "500"

2 minipaper_Class20 "200"

3 minipaper_Class16 "100"

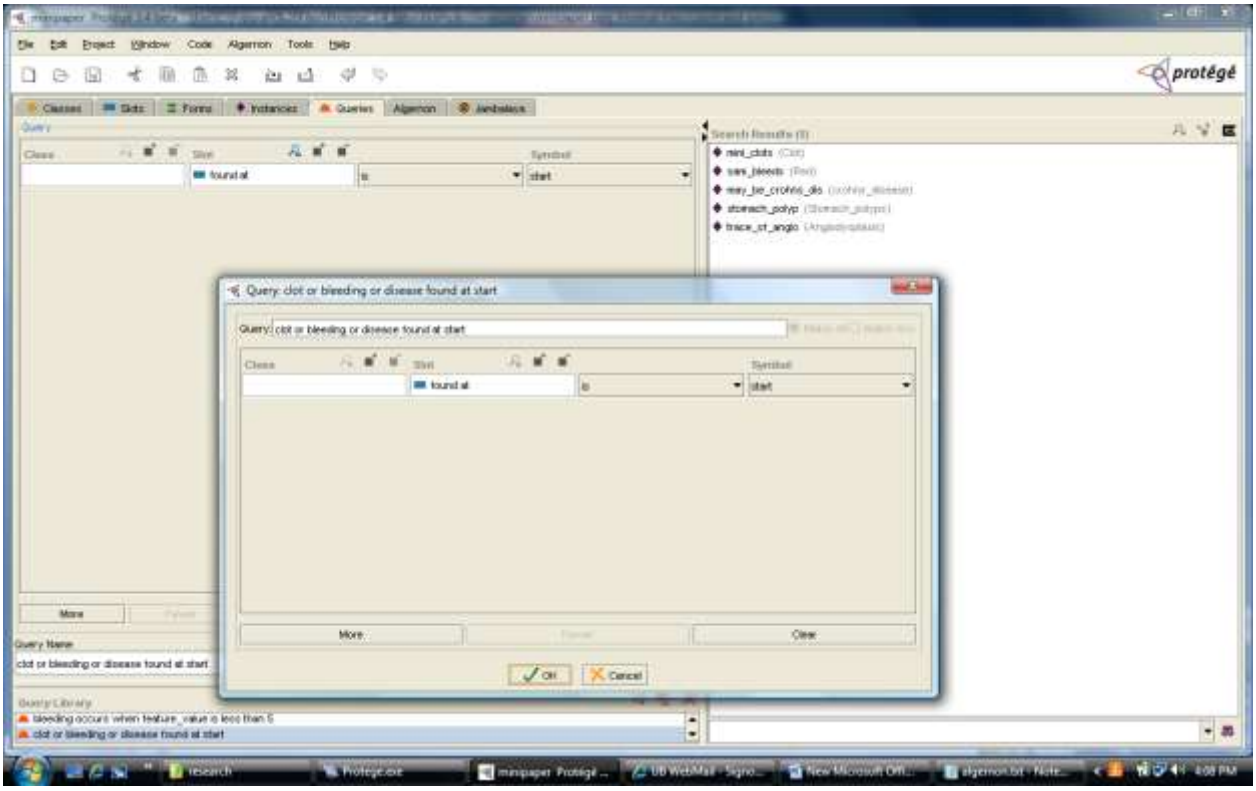


Java: Algernon output:



Query tab:

Input: clot or bleeding or disease found at start



Algernon supports forward and backward chaining rules, much like CLIPS or JESS. However, it is easier to use with Protege because it operates directly on Protege knowledge bases rather than requiring a mapping operation to and from a separate memory space. It is written in Java and is compatible with Protege v1.8, v1.9, v2.0, v2.1 and Java versions 1.3 and 1.4.

The Algernon tab allows you to execute Algernon queries and assertions within the Protege GUI. You can perform any Ask or Tell operation, including rule definition, slot value set, get and delete, frame deletion, ontology traversal, and more. It allows nearly complete access to the Protege API, as well as the ability to call external Java methods and to call an internal LISP subsystem.

Algernon through load file:

*;;; ****** WCE ******

;;;*

;;; This example illustrates the use of Algernon to build a very simple

;;; expert system. In particular it illustrates the use of the :ask

;;; form to query the user.

```
;;;
;;; Expected results:
;;;   color > 20 and weight_loss:   sam has bleeding and crohns_disease
;;;   color >15 and nausea :  sam has bleeding and stomach_polyps
;;;   Note that both answer can be true if the bleeding is
;;;   above 15 and sam has bleeding, crohns disease and stomach polyps
;;; ----------------------------------------------------------
;; Create a new KB.
(tell ((:USE-KB :WCE-EXAMPLE-KB Protege )))
;; Declare basic facts
(tell ((:TRACE :NORMAL)))
;; Define the basic taxonomy
(tell ((:taxonomy (:THING
                (Objects
                 (Symptoms reduced_appetite abdominal_pain crohns_disease stomach_polyps)
                     (Physical-Objects
                  (Patient)
                          (Diseases crohns_disease stomach_polyps)))))))
;; Four new slots:
;;  (has-disease x d) = x has disease d.
;;  (has-symptom x s) = x has symptom s.
  ;;  (color x c) = x has color c.
;;  (symptom d s)     = disease d causes symptom s.
 (tell ((:add-relation has-disease (Patient   Diseases) :max-cardinality :ANY)
```

```
(:add-relation has-symptom (Patient  Symptoms) :max-cardinality :ANY)
(:add-relation symptom     (Diseases Symptoms) :max-cardinality :ANY)
(:add-relation color (Physical-Objects :integer) :max-cardinality 1)
))
;; Backward-chaining rules to do the basic fact-finding.
(tell ((:add-rule Patient
((has-symptom ?p reduced_appetite)     <- (color ?p ?c) (:test (:LISP (> ?c 20))))  ;; Rule1
((has-symptom ?p abdominal_pain) <- (color ?p ?c) (:test (:LISP (> ?c 15))))  ;; Rule2
((has-symptom ?p weight_loss)    <- (:ask (has-symptom ?p weight_loss)))    ;; Rule3
((has-symptom ?p nausea)   <- (:ask (has-symptom ?p nausea)))            ;; Rule4
((color ?p ?c)       <- (:ask (color ?p ?c)))                ;; Rule5
)))
;; Backward-chaining rules for diagnosis
;; A person has a disease if they have all symptoms of the disease ...
;; Rule6 and Rule7
;; This version of the rules orders the queries so the
;; user is not asked unnecessary questions.
;; For example, if the patient does not have a nausea,
;; the user will not be asked if they have weight_loss.
(tell ((:add-rule Patient
((has-disease ?x crohns_disease)    <-  (has-symptom ?x reduced_appetite)     (has-symptom ?x
weight_loss))
((has-disease ?x stomach_polyps)  <-  (has-symptom ?x abdominal_pain) (has-symptom ?x nausea))
)))
```

;; Start the process

(ask ((:the-instance (?s patient) (:NAME ?s "sam"))

(:trace :normal)

(has-disease ?s ?d)

(:PRINTLN ?s " has the " ?d)

))

Source file downloaded form:
Jar file:

SimpleAlgernonExample.java

```
/*
 * Algernon - a rule-based inference engine in Java.
 * http://algernon-j.sourceforge.net/
 *
 * This example shows how to open a Protege knowledge base
 * in Java and use Algernon to query the KB.
 *
 * To run it, be sure to change the path in NEWSPAPER_PROJECT
 * to match the correct project on your system.
 */
```

```
package test_wce;
import java.util.*;
import java.io.*;

import org.algernon.Algernon;
import org.algernon.util.ErrorSet;
import org.algernon.datatype.Result;
import org.algernon.datatype.BindingList;
import org.algernon.kb.okbc.protege.AlgernonProtegeKB;
import org.algernon.kb.AlgernonKB;

/*
 * Opens the Protege newspaper KB and makes some simple queries on it.
 *
 * Example:
 *  * java  -classpath algernon.jar:protege.jar
org.algernon.test.SimpleAlgernonExample
 */
```

```
public class SimpleAlgernonExample extends Object
{
  //public static String  NEWSPAPER_PROJECT = "C:/Program
Files/Protege_3.4_beta/examples/newspaper.pprj";

  public static String  NEWSPAPER_PROJECT = "C:\\Program
Files\\Protege_3.4_beta\\examples\\newspaper\\minipaper.pprj";
  protected Algernon   f_algy       = null;   // Algernon instance
  protected AlgernonKB f_kb         = null;
  protected String     f_projectFile = "";

  public SimpleAlgernonExample()
  {
    f_projectFile = NEWSPAPER_PROJECT;
  }

  public SimpleAlgernonExample(String project)
  {
    f_projectFile = project;
  }

  /**
   * Opens the KB and initializes Algernon.
   */
  public void init()
  {
    try {
      f_algy = new Algernon();
      f_kb   = new AlgernonProtegeKB(f_algy, f_projectFile);
      f_algy.addKB(f_kb);
    }
    catch (Exception e) {
        e.printStackTrace(System.out);
      System.err.println(e + "\nUnable to open the '" + f_projectFile + "' KB.");
      stop();
    }

  }

   /**
    * Halts the program nicely.
    */
   public void stop()
   {
     if (f_kb != null)
       f_kb.close();
   }
```

```
  public static void main(String[] args)
  {
     File file = new File(SimpleAlgernonExample.NEWSPAPER_PROJECT);
     if (file.exists())
        System.out.println("##### file found.");
     else
        System.out.println("##### file NOT found.");
    SimpleAlgernonExample example = new
SimpleAlgernonExample(NEWSPAPER_PROJECT);
    example.init();
    example.start();
  }

  public void start()
  {

    try {
     // Example that retrieves all bindings of the variable ?X.
     // Note that the variable names are case-sensitive:

     ErrorSet errors = new ErrorSet();
     //---test_original----------String   query  = "((:instance Bleeding ?a)(ID ?a
?number))";
      String   query  = "((:instance Bleeding ?a)(ID ?a ?number))";

     Result result = (Result)f_algy.ask(query, errors);

     if (result == null)
     {
      System.err.println("Errors during ask of '" + query + "'");
      System.err.println(errors.toString());
     }
     else
     {
      // One way to access the results is to iterate on the results
      // using result.iterator().  Each element of the iterator
      // is of type BindingList.
      for (Iterator iterator = result.iterator();
           iterator.hasNext();)
      {
       // Result contains some BindingList objects.
       BindingList bl = (BindingList) iterator.next();

       // The bound object will either be a Java Object (Integer, Float, etc.)
       // or a KB object (typically a Protege Instance).
       // If you want Algernon objects, use algy.getAlgernonBinding().
```

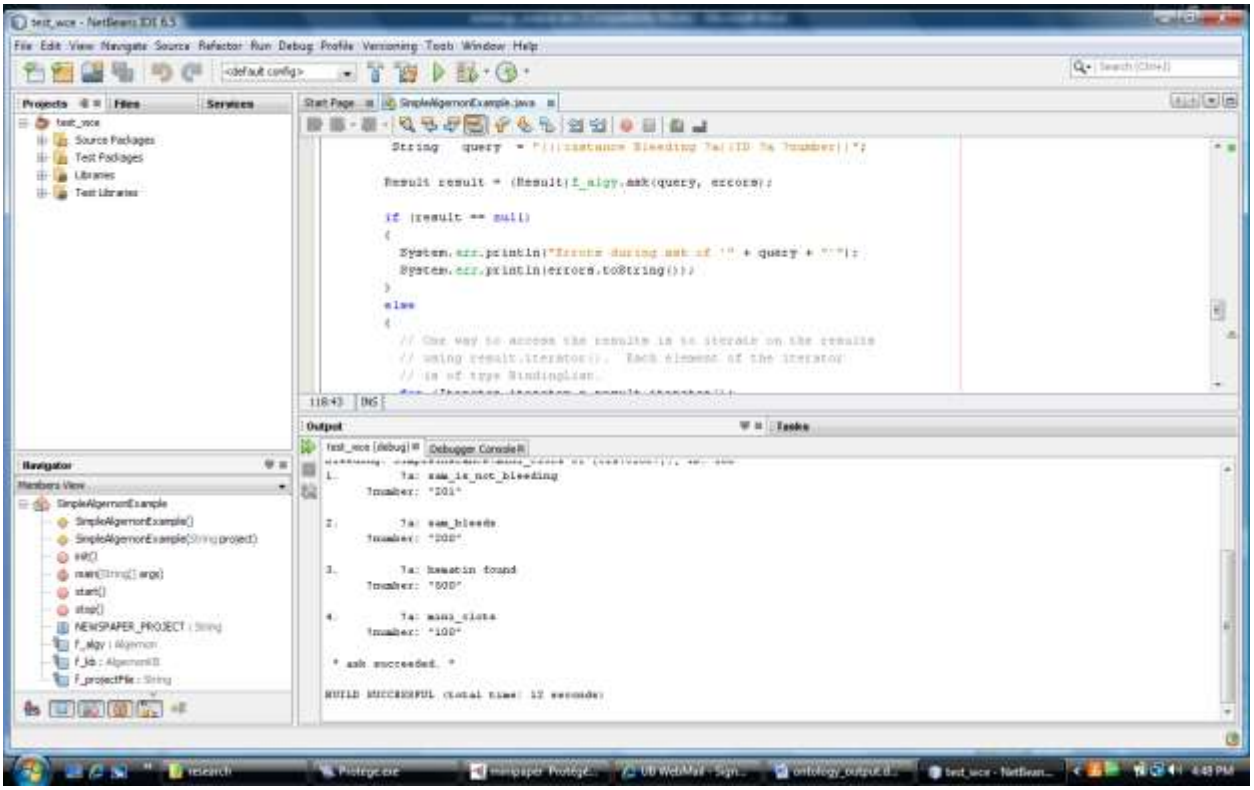
```
        Object bleeding = f_algy.getBinding("?a",    bl);
       //----------test_original// Object id   = f_algy.getBinding("?number", bl);
        Object id = f_algy.getBinding("?number",    bl);
       //-----------test_original// System.out.println("Bleeding: " + bleeding + ",
ID: " + id);
        System.out.println("Bleeding: " + bleeding + ", ID: " + id);
        // If for some reason you don't know the names of the variables
        // in the Binding Lists, you can iterate on the BindingList to
        // retrieve variable names and values. See the JavaDoc documentation
        // for the BindingList api.
      }

      // A simpler way to show the results is to let Algernon print the results:
      System.out.println(f_algy.printResultToString("ask", result));

      }
   } catch (Exception e) {
     System.err.println("Error running Algernon: " + e);
     e.printStackTrace();
   } finally {
     stop();
   }
  }

}
```

Output of Algernon:



Knowledgebase Printer.java:

```
/*
 * This application just prints out all of the classes in a knowledge base
 * and the direct instances of those classes.
 */

package test_wce;

        /*import java.io.*;*/
import java.util.*;
import edu.stanford.smi.protege.model.*;

public class KnowledgeBasePrinter {
   private static final String PROJECT_FILE_NAME = "C:\\Program
Files\\Protege_3.4_beta\\examples\\newspaper\\newspaper.pprj";
        public static void main(String[] args ){
        Collection errors = new ArrayList();
        Project project = new Project(PROJECT_FILE_NAME, errors);
        if (errors.size() == 0) {
            KnowledgeBase kb = project.getKnowledgeBase();
            Iterator i = kb.getClses().iterator();
            while (i.hasNext()) {
                  Cls cls = (Cls) i.next();
               System.out.println("Class: " + cls.getName());
```

```
            Iterator j = cls.getDirectInstances().iterator();
            while (j.hasNext()) {
                  Instance instance = (Instance) j.next();
               System.out.println("  Instance: " + instance.getName());
            }
         }
      } else {
         displayErrors(errors);
      }
      waitForContinue();
   }

   private static void displayErrors(Collection errors) {
       Iterator i = errors.iterator();
      while (i.hasNext()) {
         System.out.println("Error: " + i.next());
      }
   }

   private static void waitForContinue() {
      System.out.println("Press <Enter> to continue");
      try {
         System.in.read();
       } catch (Exception e) {}
   }
}
```

Disadvantages of Algernon:
No proper example provided. I tried using "mini mycin" as an example but it couldn't create instances and query properly. I switched to Jess which is another type of inference engine. I have given the reasons as why Jess is good compared to other inference engines like Algernon, SweetRules .

Jess:

- Java Expert System Shell; based on CLIPS
- Forward chaining; production rules
- Fact-base and pattern matching
- Lisp-like syntax

Jess Rule Engine:
The Jess system consists of a rule base, a fact base, and an execution engine. The execution engine matches facts in the fact base with rules in the rule base. These rules can assert new facts and put them in the fact base or execute Java functions.

I chose Jess because it works seamlessly with Java, has an extensive user base, is well documented, and is very easy to use and configure. Jess provides both an

interactive command line interface and a Javabased API to its rule engine. This engine can be embedded in Java applications and provides a flexible two-way runtime communication between Jess rules and Java.

Reasoning in Jess is based on a list of known facts and a set of rules that try to match on these facts in its fact base.

Facts are basically a list of known values, such as (temperature outdoor 25), which represents the value 25 for the temperature at the outdoor location. The rules' preconditions are patterns with wild cards and conditional expressions, which match such facts. For example, the pattern (temperature ? ?t&:(> ?t 20)) matches temperatures greater than 20, while ignoring the location. Rules can assert new facts when they fire, which in turn could result in other rules firing. Jess uses the Rete algorithm to ensure efficient pattern matching and rule activation. Jess supports mainly forward chaining and, to some extent, backward chaining.

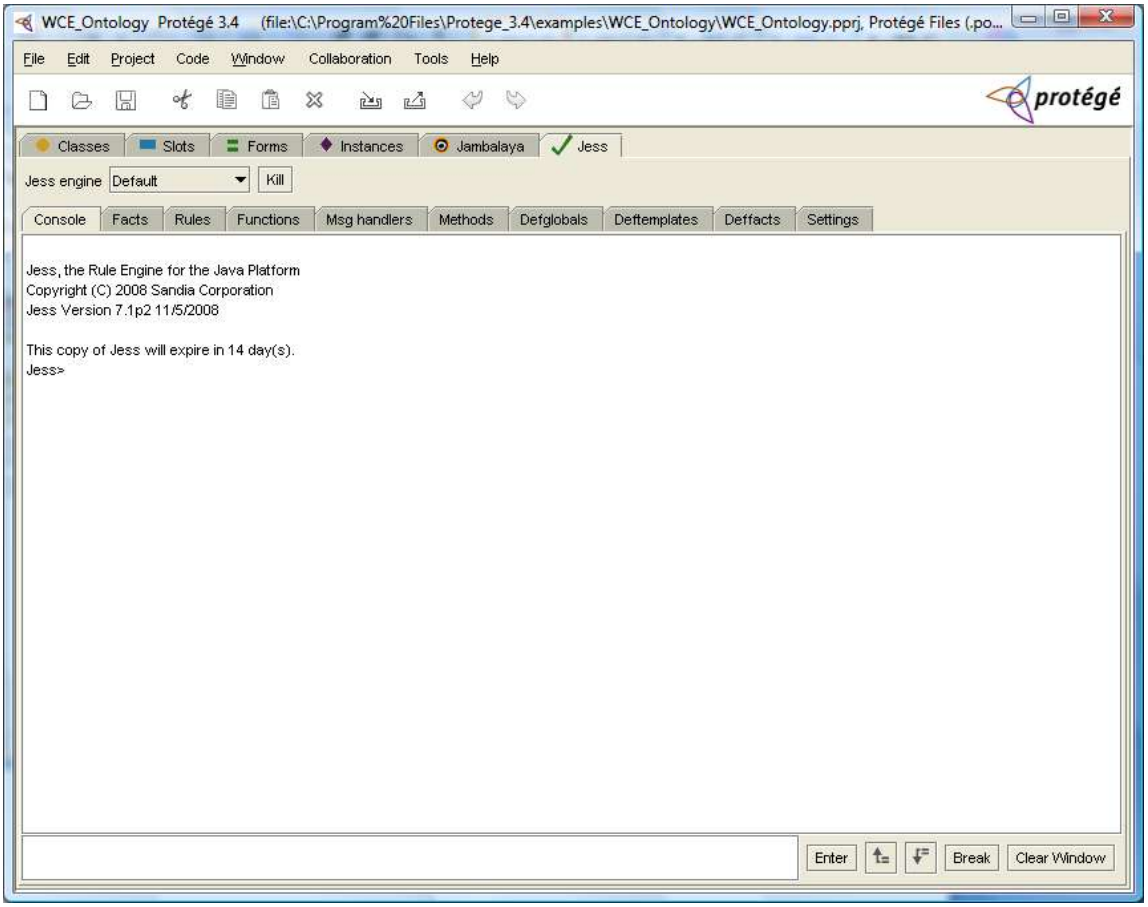
Jess version used: 7.1p2

Where to download:
http://www.jessrules.com/jess/download.shtml
Kindly unzip the files.

Where to copy the downloaded jar file:
Go to the downloaded jess files. Goto lib folder in that there are two jar files "jess.jar" and "jsr94.jar" copy those in to your protégé:

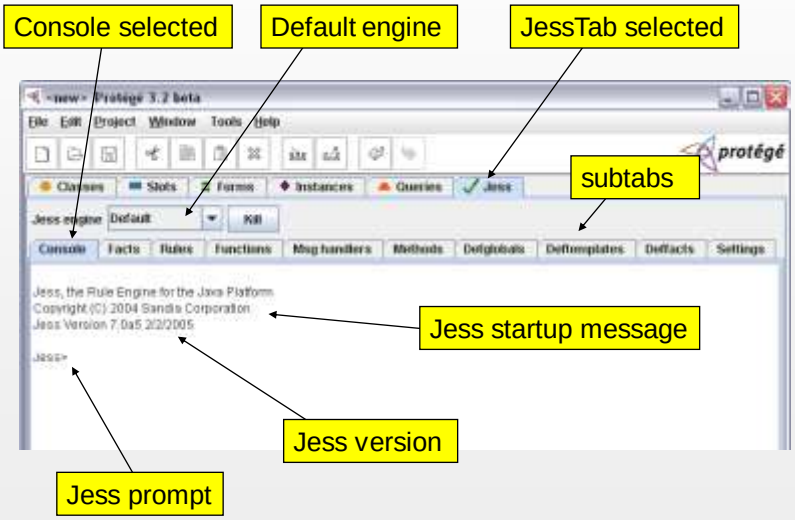
Where in protégé: go to your protégé 3.4 file location
C:\Program Files\Protege_3.4\jre\lib
Paste the files and in "plugin" folders
C:\Program Files\Protege_3.4\plugins\se.liu.ida.JessTab
And
C:\Program Files\Protege_3.4\plugins\edu.stanford.smi.protegex.owl

Go to protégé →project→configure→check Jess tab



What does each and every tab mean:

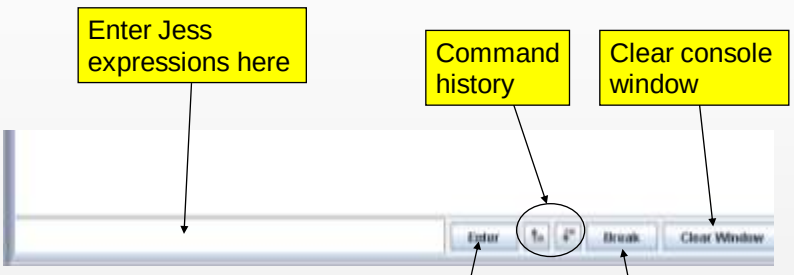
Console window (upper part)



Linköpings universitet

JessTab Tutorial 2006 20

Console window (lower part)



Linköpings universitet

JessTab Tutorial 2006 21

Basic commands in Jess:

(run) : To invoke the rules

(exit) :Exits Protégé if running in Jess Tab

(facts) :To display facts

(rules): To display rules

(printout t : output stream (t = stdout)

crlf :New line

(slot-set john age 21) : Sets the slot value

(slot-get john age) : gets the slot value

(make-instance sue of Person (name "Sue") (age 22)) :To create instance

(mapclass <class-name>) or (mapinstance <instance>) :To map instances to facts

(defrule …)Define the rules matching the object facts

To define methods:

(defmethod <name> [<index>] [<comment>]

(<parameter-restriction>* [<wildcard-parameter-restriction>])

<action>*

To define Message handler:

(defmessage-handler <class-name> <message-name>

[<handler-type>] [<comment>]

(<parameter>* [<wildcard-parameter>])

<action>*)

To invoke message handler:

(send <instance> <message> <param>*)

The general format of *deftemplate* is

(deftemplate <relation-name> [<optional-comment>]
<slot-definition>*)

where <slot-definition> is defined as

(slot <slot-name>) | (multislot <slot-name>)

Multifield slots:

- Normal, single-field slots can hold only one value
- *Multifield* slots can hold several values
- The values are ordered in a list

Add new facts with the assert command: (assert <fact>+)
More than one fact can be added through assert

Functions for Knowledge base operations:

mapclass
mapinstance
unmapinstance
defclass
make-instance
initialize-instance
modify-instance
duplicate-instance
definstances
unmake-instance
slot-get
slot-set
slot-replace$
slot-insert$
slot-delete$
slot-facets
slot-types
slot-cardinality

slot-range
slot-allowed-values
slot-allowed-classes
slot-allowed-parents
slot-documentation
slot-sources
facet-get
facet-set
class
class-existp
class-abstractp
class-reactivep
superclassp
subclassp
class-superclasses
class-subclasses
get-defclass-list
class-slots

instancep
instance-existp
instance-name
instance-address
instance-addressp
instance-namep
slot-existp
slot-default-value
set-kb-save
get-kb-save
load-kb-definitions
load-project
include-project
save-project
jesstab-version-number
jesstab-version-string
get-knowledge-base
get-tabs

Why forward and backward chaining:
If all (relevant) facts are already known, and the purpose of the system is to find where that information leads, forward-chaining should be selected. If, on the other hand, few or no facts are known and the goal is to find if one of many possible conclusions is true,
use backward-chaining.

Jess is a rule engine. In the simplest terms, this means that Jess's purpose it to continuously apply a set of if-then statements (*rules*) to a set of data (the *working memory*). You define the rules that make up your own particular rule-based system. Jess rules look something like this:

```
Jess> (defrule library-rule-1
    (book (name ?X) (status late) (borrower ?Y))
    (borrower (name ?Y) (address ?Z))
   =>
    (send-late-notice ?X ?Y ?Z))
```

This rule might be translated into pseudo-English as follows:

```
Library rule #1:
If
  a late book exists, with name X, borrowed by someone named Y
and
  that borrower's address is known to be Z
then
  send a late notice to Y at Z about the book X.
```

RULES:

- Put the *most specific* patterns near the top of each rule's LHS.
- Put the patterns that will match the *fewest facts* near the top of each rule's LHS.
- Put the *most transient* patterns (those that will match facts that are frequently retracted and asserted) near the bottom of a LHS.

You can use the `view` command to find out how many partial matches your rules generate.

(defrule rule-name
"optional comment"
(pattern-1) ; left-hand side (LHS) of the rule
(pattern-2) ; consisting of elements before the "=>"
(pattern-n)
=>
(action-1) ; right-hand side (RHS) of the rule
(action-2) ; consisting of elements after the "=>"
(action-m)
) ; the last ")" balances the opening "(" to
; the left of "defrule". Be sure all your
; parentheses balance or you will get
; error messages.

The entire rule must be surrounded by parentheses. Only one rule name can exist at one time in Jess. An action is actually a function which typically has no return value, but performs some useful action, such as an assert or retract. A rule often has multiple patterns and actions. Zero or more patterns may be written after the rule header. Each pattern consists of one or more fields. The symbol => that follows the patterns in a rule is called an arrow. The arrow represents the beginning of the THEN part of an IF-THEN rule

Why LISP(List Processing): can implement complex logical relationships with very little code. The Jess language is a highly specialized form of Lisp.

Short description about Jess rule engine:

Jess is a *rule engine* for the Java platform. To use it, you specify logic in the form of rules using one of two formats: the Jess rule language (prefered) or XML. You also

provide some of your own data for the rules to operate on. When you run the rule engine, your rules are carried out. Rules can create new data, or they can do anything that the Java programming language can do.

Although Jess can run as a standalone program, usually you will embed the Jess library in your Java code and manipulate it using its own Java API or the basic facilities offered by the javax.rules API.

You can develop Jess language code in any text editor, but Jess comes with a full featured development environment based on the award-winning Eclipse platform.

Representing OWL Concepts as Jess Knowledge

Relevant knowledge about OWL individuals must be represented as Jess knowledge. The two primary properties that must be represented are
1) the classes to which an individual belongs and
2) the properties the individual possesses.
Same-as and different-from information about these individuals must also be captured.
The Jess template facility provides a mechanism for representing an OWL class hierarchy. A Jess template hierarchy can be used to model a class hierarchy using a Jess slot to hold the name of the individual belonging to the hierarchy.
For Example: A user must define a Jess template to represent the owl:Thing class:
(deftemplate OWLThing (slot name))

A hierarchy representing a class Red that subclasses Bleeding that again subclasses Bleeding_Event and it subclasses from Abnormalities which is a subclass from Bleeding_Event a direct subclass of owl:Thing called Bleeding_Event could then be represented as follows in Jess:
(deftemplate Bleeding_Event extends OWLThing)
(deftemplate Abnormalities extends Bleeding_Event)
(deftemplate Bleeding extends Bleeding_Event)
(deftemplate Red extends Bleeding_Event)

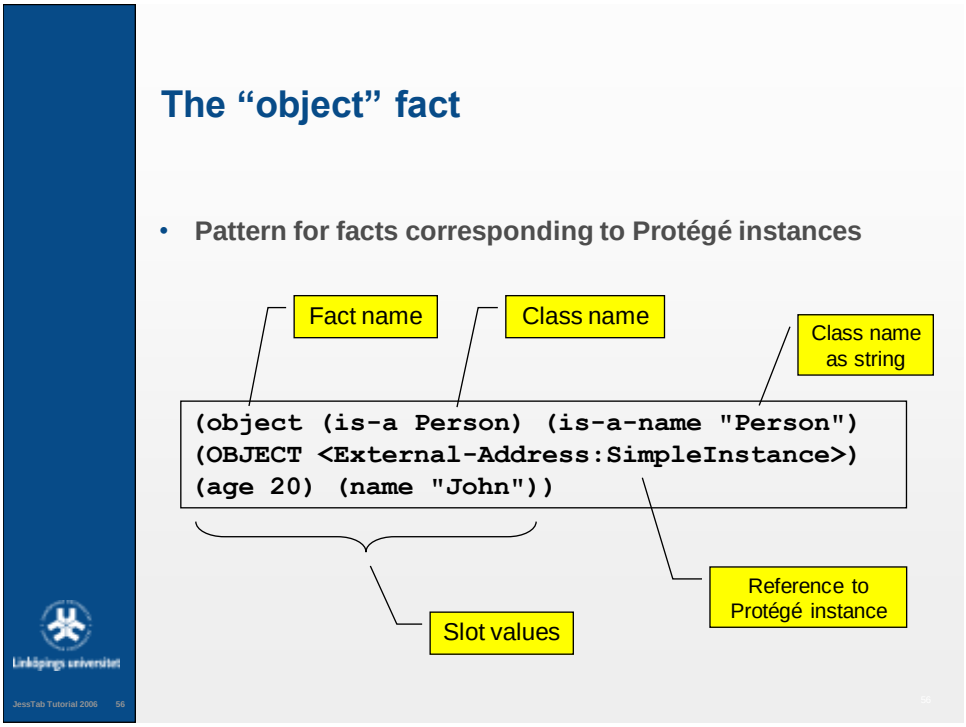
Using this template definition, the OWL individual can be asserted as a member of the class Red:
(assert (Red (name sam_bleeds))

OWL property information can be directly represented as Jess facts.
For example, the information that an individual
Fred is related to individual Joe through the hasUncle property can be directly asserted using:
(assert (hasUncle Fred Joe))
Similarly, OWL's same-as and different-from relationships between individuals can be directly represented in Jess.
For example, the information that Fred and Frederick are the same OWL individual can be expressed:

(assert (sameAs Fred Frederick))

Mapping Protégé knowledge bases to Jess facts

For example, mapping the class Person, which has a single instance representing John, age 20, results in a Jess fact with the format (object (is-a Person) (name “John”) (age 20)). This fact represents the instance with its slot values for name and age.



JessTab asserts facts based on a certain pattern (template). The pattern starts with object and always contains the class name and a direct reference to the Protégé object. As the example shows, the instance’s slot–value pairs are also part of the fact. The mapping mechanism asserts a fact for every instance (of a mapped class). Creating a new instance of the class Person, for example, will result in the assertion of a new fact matching the instance.

we can write rules that match instances regardless of their class by using a wild card for the is-a property. The pattern (object (is-a ?) (name “John”)) matches every (mapped) instance that has a slot name with the value “John.” (The character ? represents a wild card in patterns.)

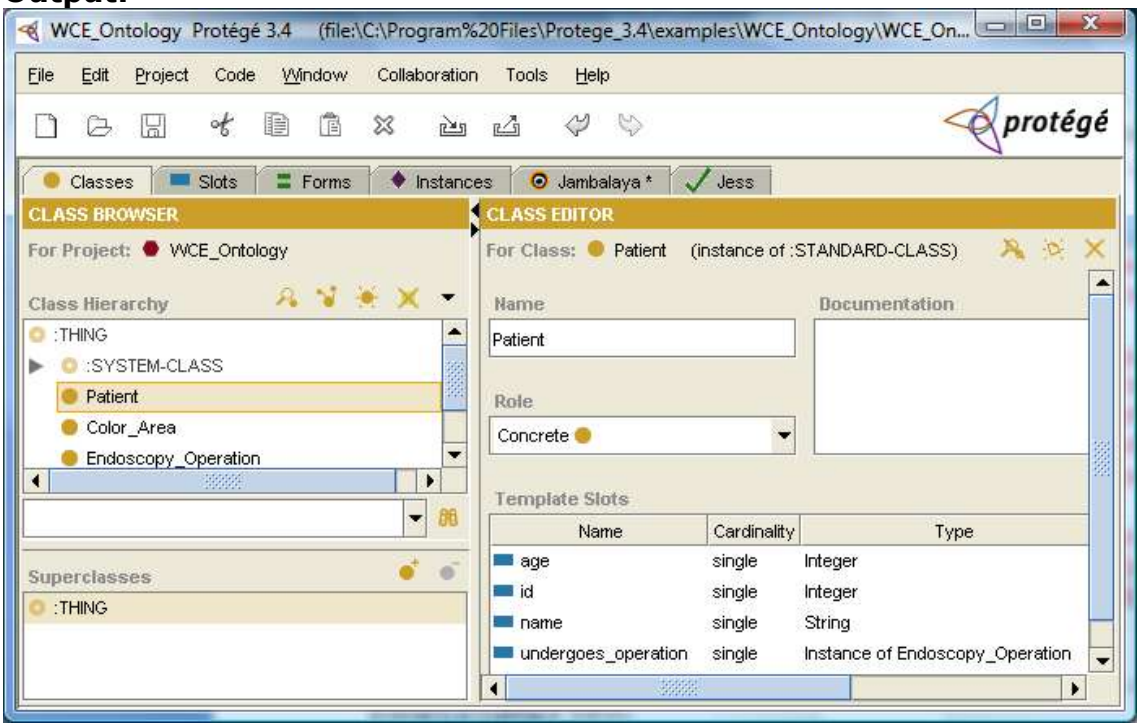
The following example illustrates how we create a Patient class and a patient instance named John:

```
Jess>  (defclass Patient (is-a :THING)
(slot name (type string))
(slot age (type integer))
(slot id (type integer))
```

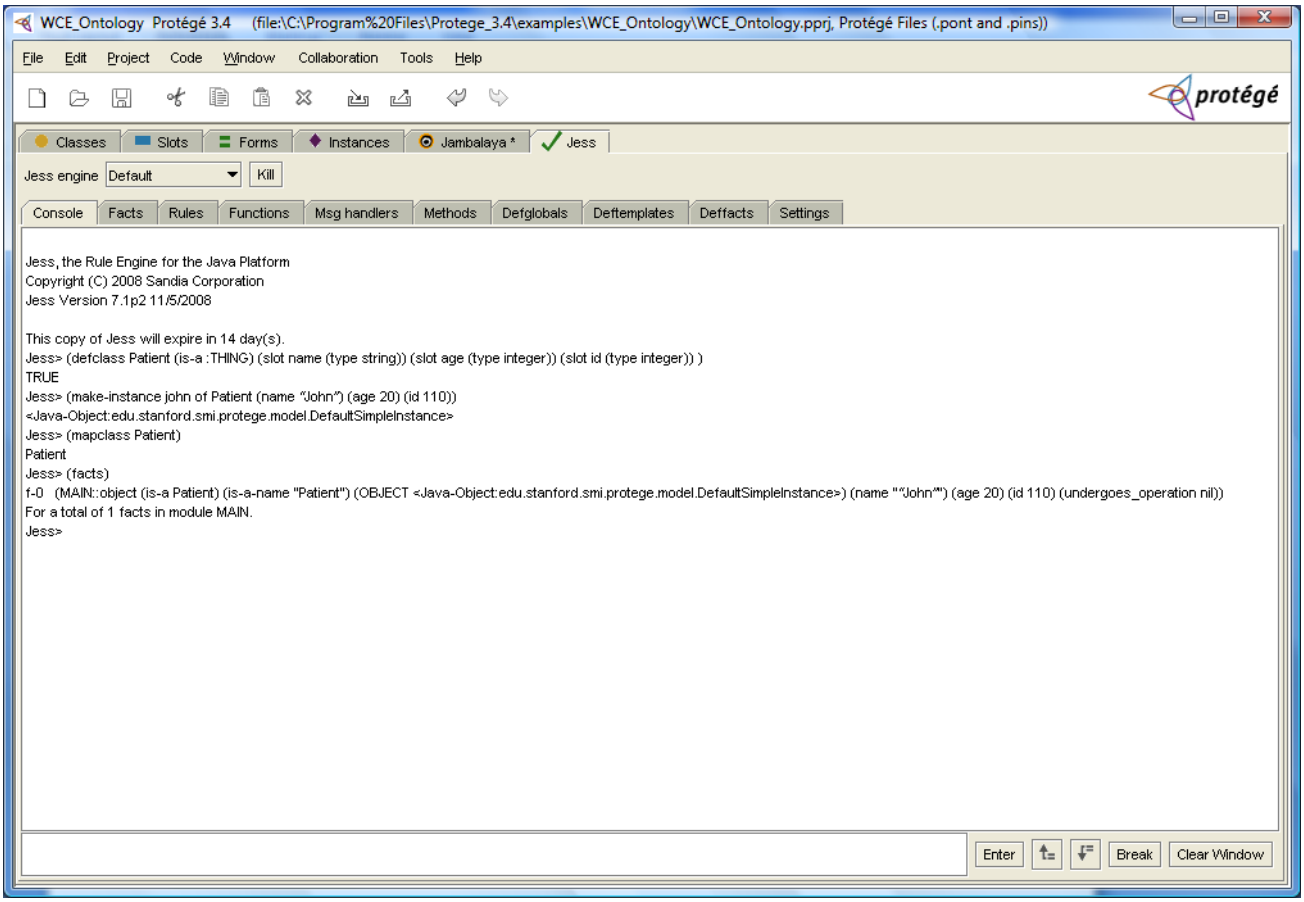
)
TRUE
Jess> (make-instance john of Patient (name “John”)
(age 20) (id 110))
Jess> (mapclass Patient)
Patient

The mapclass expression marks the Patient class as mapped, and it will assert facts of the instances (direct and indirect) of Patient. After the mapclass operation, we can check the resulting Jess fact:
Jess> (facts)
f-0 (MAIN::object (is-a Patient) (is-a-name "Patient") (OBJECT <Java-Object:edu.stanford.smi.protege.model.DefaultSimpleInstance>) (name ""John"") (age 20) (id 110) (undergoes_operation nil))
For a total of 1 facts in module MAIN.

Output:



In jess:



Example: We use the slot-set function to change the age to 20:

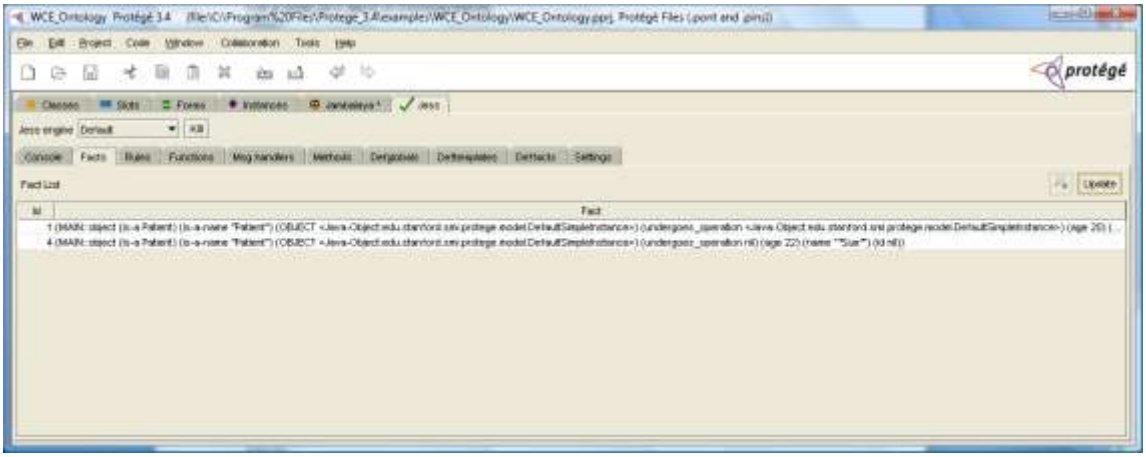
Jess> (slot-set john age 20)
Jess> (facts)
f-1 (MAIN::object (is-a Patient) (is-a-name "Patient") (OBJECT <Java-Object:edu.stanford.smi.protege.model.DefaultSimpleInstance>) (undergoes_operation <Java-Object:edu.stanford.smi.protege.model.DefaultSimpleInstance>) (age 20) (name ""John"") (id 110))
For a total of 1 facts in module MAIN.

Creating a second instance adds a new fact to the Jess list of facts:

Jess> (make-instance sue of Patient (name "Sue") (age 22))
<Java-Object:edu.stanford.smi.protege.model.DefaultSimpleInstance>
Jess> (facts)
f-1 (MAIN::object (is-a Patient) (is-a-name "Patient") (OBJECT <Java-Object:edu.stanford.smi.protege.model.DefaultSimpleInstance>) (undergoes_operation <Java-Object:edu.stanford.smi.protege.model.DefaultSimpleInstance>) (age 20) (name ""John"") (id 110))

f-4 (MAIN::object (is-a Patient) (is-a-name "Patient") (OBJECT <Java-Object:edu.stanford.smi.protege.model.DefaultSimpleInstance>) (undergoes_operation nil) (age 22) (name ""Sue"") (id nil))
For a total of 2 facts in module MAIN.

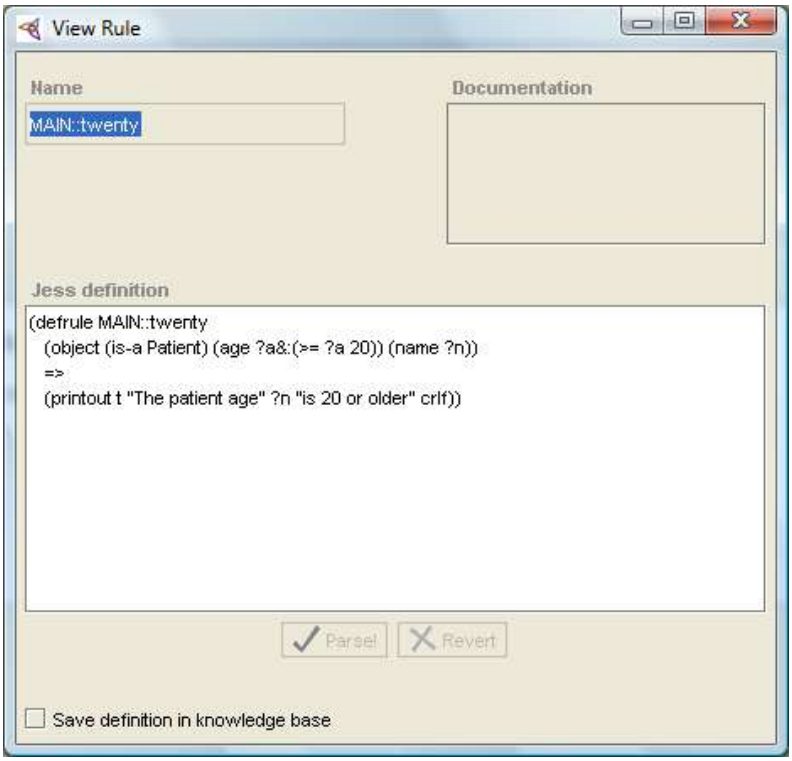
Output in Jess/ facts tab:



We can now write a Jess rule that pattern matches on the facts. The following rule prints the name for every patient age 20 or older:

Jess> (defrule twenty
(object (is-a Patient) (name ?n) (age ?a&:
(>= ?a 20)))
=>
(printout t "The patient age" ?n "is 20 or older" crlf))
TRUE
Jess> (run)
The patient age"Sue"is 20 or older
The patient age"John"is 20 or older
2

Output: Jess/rules tab



To actively modify the knowledge base, we need functions for accessing and modifying values in Protégé. The functions **slot-get** and **slot-set** let us retrieve and change slot values, respectively.

Message handlers and methods

You can use the defmessage-handler and defmethod constructs to create object-oriented programs based on Protégé ontologies.

For example, the message handler lets us send messages to instances of the Protégé class Patient.

```
(defmessage-handler Patient incrementAge (?i)
(slot-set ?self age (+ (slot-get ?self age) ?i)))
```

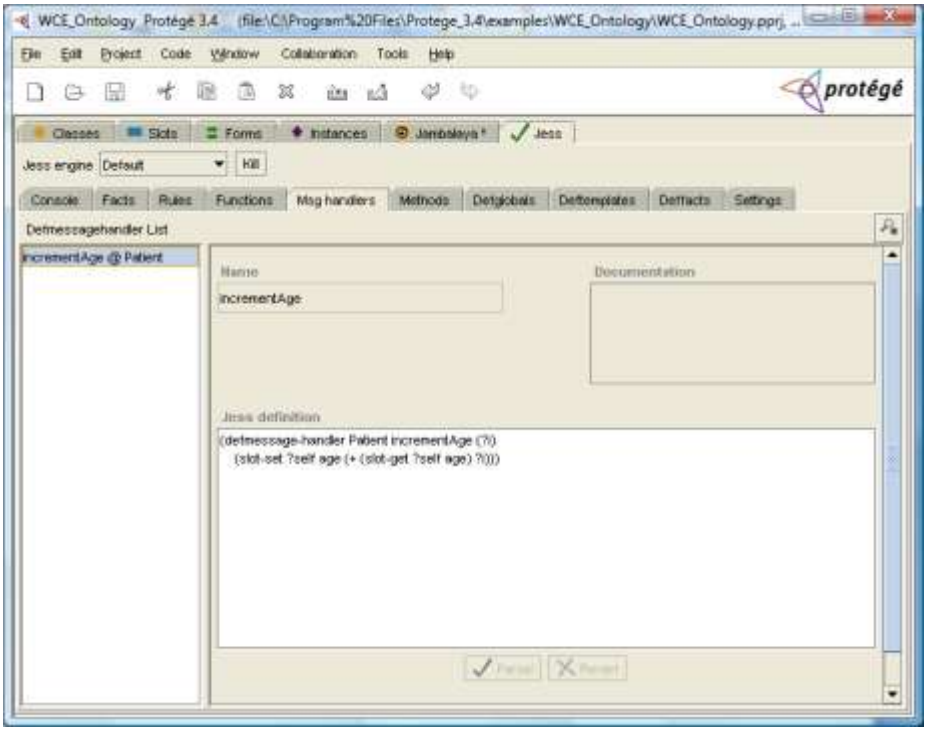
The expression

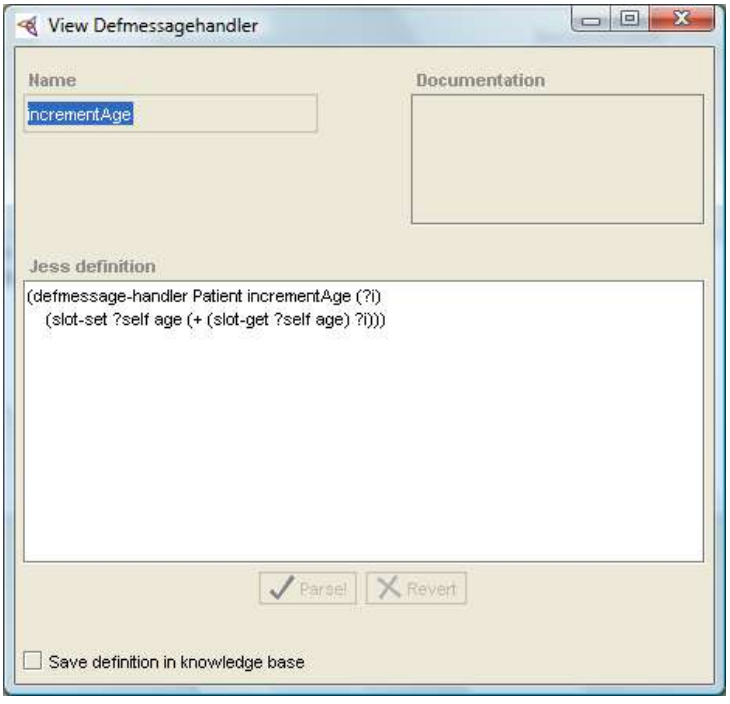
```
(send john incrementAge 1)
```

sends the incrementAge message to the john instance. In this case, the message handler then adds 1 to the value of the slot age of the instance john.

```
Jess> (facts)
f-4   (MAIN::object (is-a Patient) (is-a-name "Patient") (OBJECT <Java-
Object:edu.stanford.smi.protege.model.DefaultSimpleInstance>)
(undergoes_operation nil) (age 22) (name ""Sue"") (id nil))
f-5   (MAIN::object (is-a Patient) (is-a-name "Patient") (OBJECT <Java-
Object:edu.stanford.smi.protege.model.DefaultSimpleInstance>)
(undergoes_operation <Java-
Object:edu.stanford.smi.protege.model.DefaultSimpleInstance>) (age 21) (name
""John"") (id 110))
```

For a total of two facts in module MAIN.
Output: jess/msghandler tab:



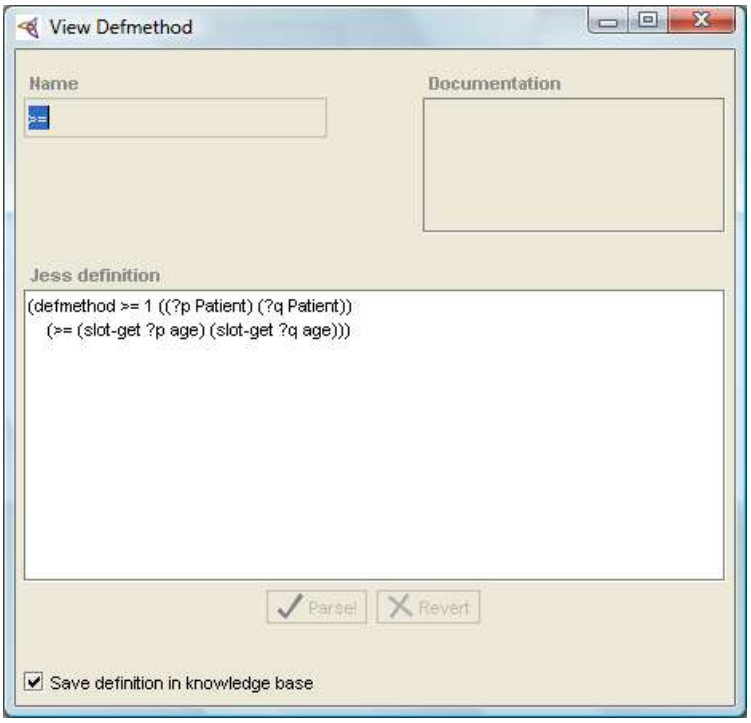


Methods are similar to message handlers in that they let us define behavior for instances. However, the method dispatcher can consider several parameters, and the methods can overload existing functions.

For example, the method:

(defmethod >= ((?p Patient) (?q Patient))
(>= (slot-get ?p age) (slot-get ?q age)))

overloads the built-in Jess function >= and lets us compare Person instances on the basis of their age. You can use expressions such as (>= john sue) to compare patient's ages. Hence, the message handlers and methods complement the mapping of instances to facts by providing elements for object-oriented programming.



Jess Tab lets us use Jess as a programming language and inference engine and it is used to implement operations on the knowledge base. Because Jess is Turing complete, we can use it to implement custom-tailored constraint checkers and programs that modify the knowledge base.

Defining Functions in Jess

You can define your own functions in the Jess rule language using the `deffunction` construct. A `deffunction` construct looks like this:

```
(deffunction <function-name> [<doc-comment>] (<parameter>*)

   <expr>* [<return-specifier>])
```

The <function-name> must be a symbol. Each <parameter> must be a variable name. The optional <doc-comment> is a double-quoted string that can describe the purpose of the function. There may be an arbitrary number of <expr> expressions. The optional <return-specifier> gives the return value of the function.

```
Jess> (deffunction max (?a ?b)
(if (> ?a ?b)
then
(return ?a)
else
(return ?b)))
TRUE
Or
Jess> (deffunction max (?a ?b)
(if (> ?a ?b)
then
?a
else
?b))

Jess> (printout t "The greater of 3 and 5 is " (max 3 5) "." crlf)
The greater of 3 and 5 is 5.
```

The general syntax of a deffunction is shown following.

```
(deffunction <function-name>
(?arg1 ?arg2 ...?argM [$?argN]) ; argument list. Last one
; may be optional multifield
; argument.
[optional comment]
<action-1> ; action-1 to action-n-1 do
<action-2> ; not return a value
<action-n-1>
<action-n> ; only last action returns a
; value
)
```

The ?arg are dummy arguments, which means that the names of the arguments will not conict with variable names in a rule if they are the same. The term dummy argument is sometimes called a parameter in other books. Although each action may have returned values from function calls within the action, these are blocked by the deffunction from being returned to the user. The deffunction will only return the value of the last action, <action-n>. This action may be a function, a variable, or a constant.

Working Memory

Each Jess rule engine holds a collection of knowledge nuggets called *facts.* This collection is known as the *working memory.* Working memory is important because rules can only react to additions, deletions, and changes to working memory. You can't write a Jess rule that will react to anything else.

Every fact has a *template*. The template has a name and a set of *slots*, and each fact gets these things from its template. This is the same structure that JavaBeans -- plain old Java objects, or POJOs -- have, and it's also similar to how relational databases are set up. The template is like the class of a Java object, or like a relational database table. The slots are like the properties of the JavaBean, or the columns of a table. A fact is therefore like a single JavaBean, or like a row in a database table.

In Jess, there are three kinds of facts: *unordered facts*, *shadow facts* and *ordered facts*.
Facts are added using the assert, add, and definstance functions. You can remove facts with the retract and undefinstance functions. The modify function lets you change the slot values of facts already in working memory. And finally, you can completely clear Jess of all facts and other data using the clear command.

Templates

As we've already stated, every fact has a *template*. A fact gets its *name* and its list of *slots* from its template. Therefore a template is something like a Java class.

You usually create templates yourself using the deftemplate construct or the defclass function. Other times, templates are created automatically for you, either while you're defining a defrule or when you use the add function or the jess.Rete.add(java.lang.Object) method.

The deftemplate construct is the most general and most powerful way to create a template. You won't understand most of the options shown here yet, but we'll cover them all in this chapter and the next:

```
(deftemplate template-name
    [extends template-name]
    ["Documentation comment"]
    [(declare  (slot-specific TRUE | FALSE)
               (backchain-reactive TRUE | FALSE)
               (from-class class name)
               (include-variables TRUE | FALSE)
               (ordered TRUE | FALSE))]
     (slot | multislot slot-name
         ([(type ANY | INTEGER | FLOAT |
                NUMBER | SYMBOL | STRING |
                LEXEME | OBJECT | LONG)]
         [(default default value)]
         [(default-dynamic expression)]
         [(allowed-values expression+)])*)
```

A template declaration includes a name, an optional documentation string, an optional "extends" clause, an optional list of declarations, and a list of zero or more slot descriptions. Each slot description can optionally include a type qualifier or a default value qualifier. In the syntax diagram, defaults for various options are indicated in bold letters.

The *template-name* is the head of the facts that will be created using this template.

The declarations affect either how the template will be created, or how facts that use it will act, or both.

Undefining templates

you can remove a previously defined template using the `jess.Rete.removeDeftemplate(String)` method.

Creation of classes and instances:

1) Patient

```
Jess>
(defclass Patient (is-a :THING)
(slot name (type string))
(slot age (type integer))
(slot id (type integer))
)

Jess> (make-instance john of Patient (name “John”)
(age 20) (id 110))


(object (is-a Patient) (name “John”)
(age 20))
```

2) Endoscopy_operation

```
(defclass Endoscopy_Operation (is-a :THING)
(slot Doctor_name (type string))
(slot Hospital_name (type string))
(slot contains_video (type string))
)

Jess> (make-instance Polyp_test of Endoscopy_Operation (Doctor_name “Padmini”)
(Hospital_name st.Vincets Healthcare))
```

3) Video

```
(defclass Video (is-a :THING)
(slot video_id (type integer))
(slot frame (type integer))
(slot framerate (type integer))
(slot resolution (type integer))
(slot contains_frame (type string))
)
```

Jess> (make-instance John_video of Video (video_id “110_John”)
(framerate 5))

4) Frame
(defclass Frame (is-a :THING)
(slot contains_color_area (type string)))

(make-instance Bleeding_110_john_colorarea of Frame (contains_color_area
“Bleeding_110_john_colorarea”))
(make-instance NonBleeding_110_john_colorarea of Frame (contains_color_area
“NonBleeding_110_john_colorarea”))

5) Color_Area
(defclass Color_Area (is-a :THING)
(slot Hue (type float))
(slot saturation (type integer))
(slot brightness (type integer))
(slot boundary (type boolean))
)

Jess> (make-instance 110_frame1_area2 of Color_Area (Hue 0.90)
(saturation 87)
(brightness 45)
)

(make-instance 110_frame1_area3 of Color_Area (Hue 0.95)
(saturation 96)
(brightness 49)
)

(make-instance 110_frame2_area1 of Color_Area (Hue 0.60)
(saturation 66)
(brightness 35)
)
(make-instance 110_frame2_area2 of Color_Area (Hue 0.55)
(saturation 68)
(brightness 34)
)
(make-instance 110_frame2_area3 of Color_Area (Hue 0.83)
(saturation 22)
(brightness 24)
)

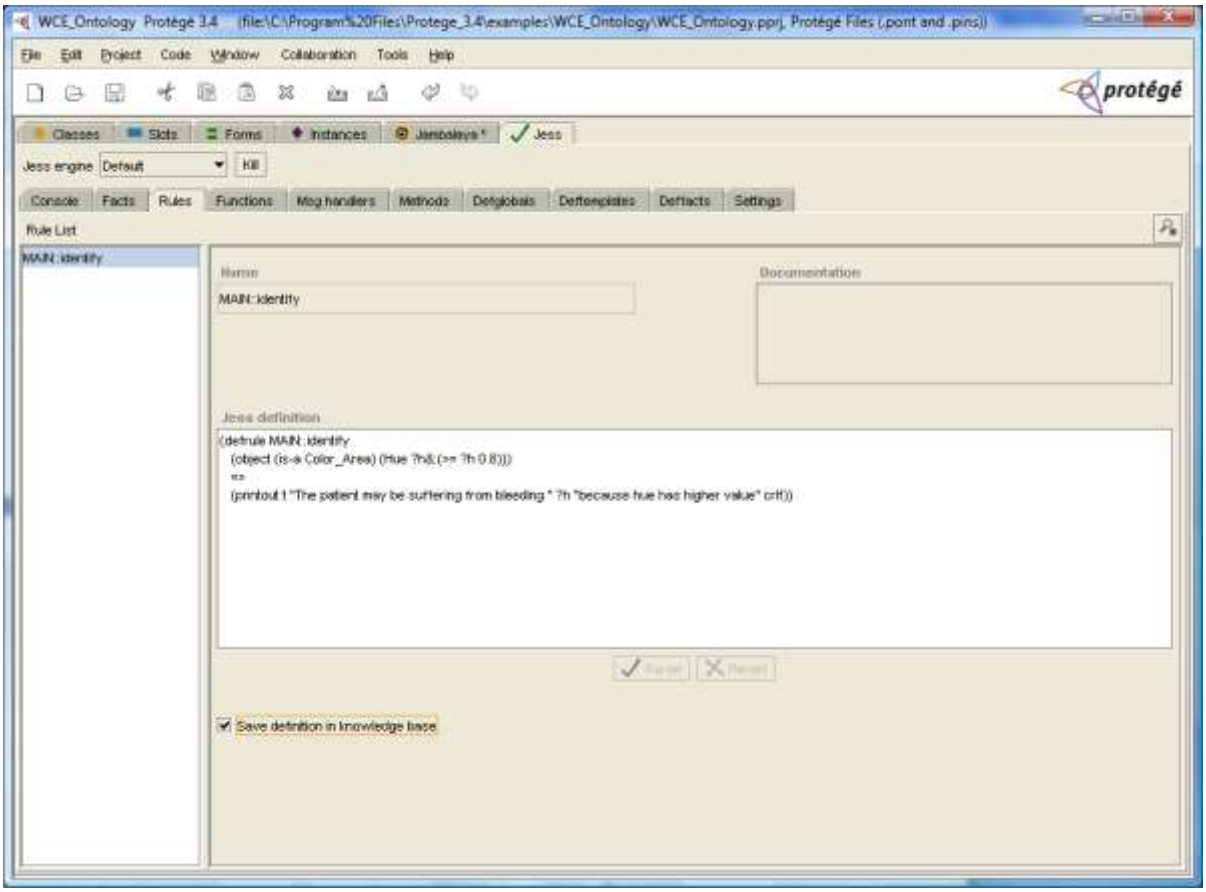
defining rule:
Jess> (defrule twenty
(object (is-a Patient) (name ?n) (age ?a&:
(>= ?a 20)))
=>
(printout t “The patient age ” ?n “ is 20 or
older” crlf))

TRUE
Jess> (run)
The person John is 21 or older
The person Sue is 21 or older
2

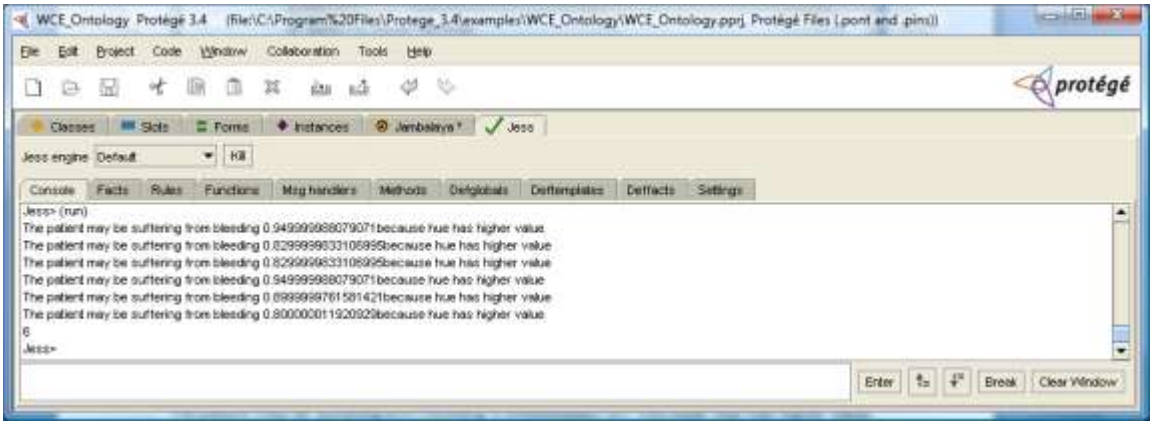
(defrule changeage
?o <- (object (is-a Patient) (age 22))
=>
(slot-set ?o age 23))

Jess rule output:
(mapclass Color_Area)
(facts)

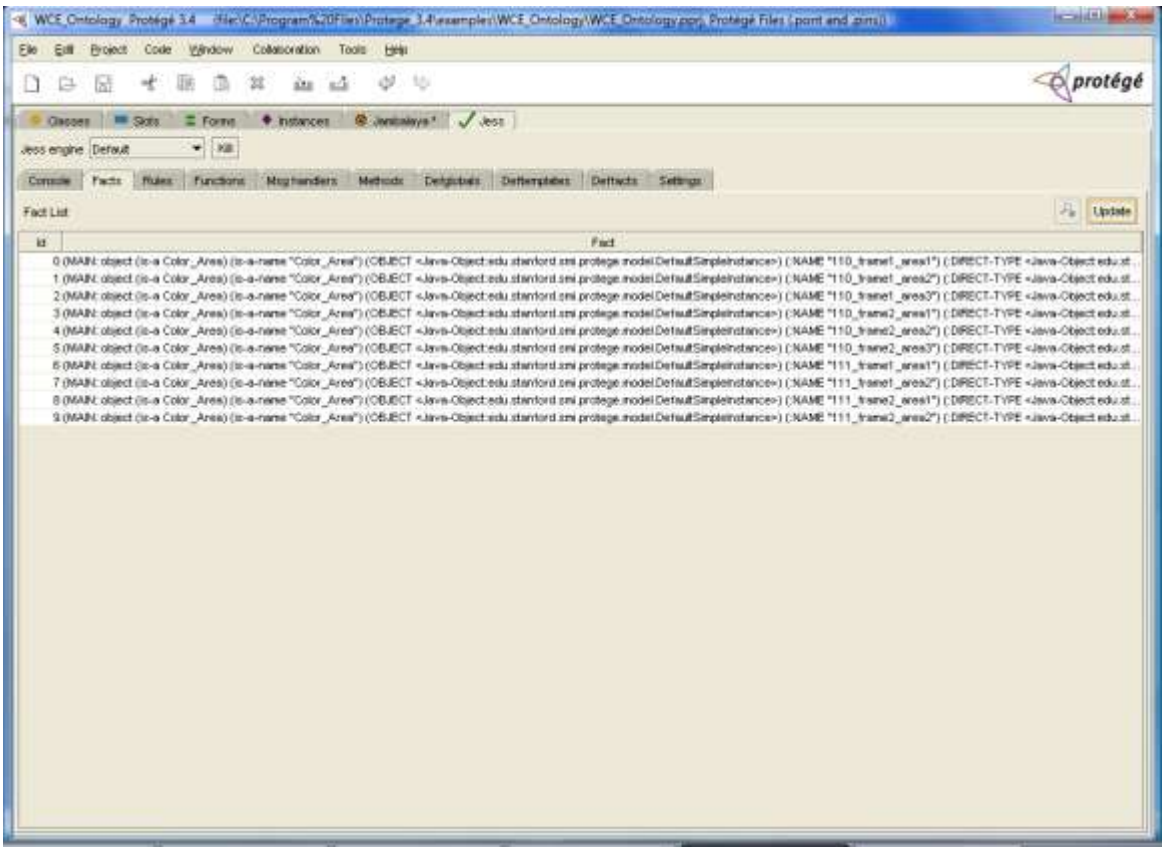
(defrule MAIN::identify
  (object (is-a Color_Area) (Hue ?h&:(>= ?h 0.8)))
  =>
  (printout t "The patient may be suffering from bleeding " ?h "because hue has higher value" crlf))



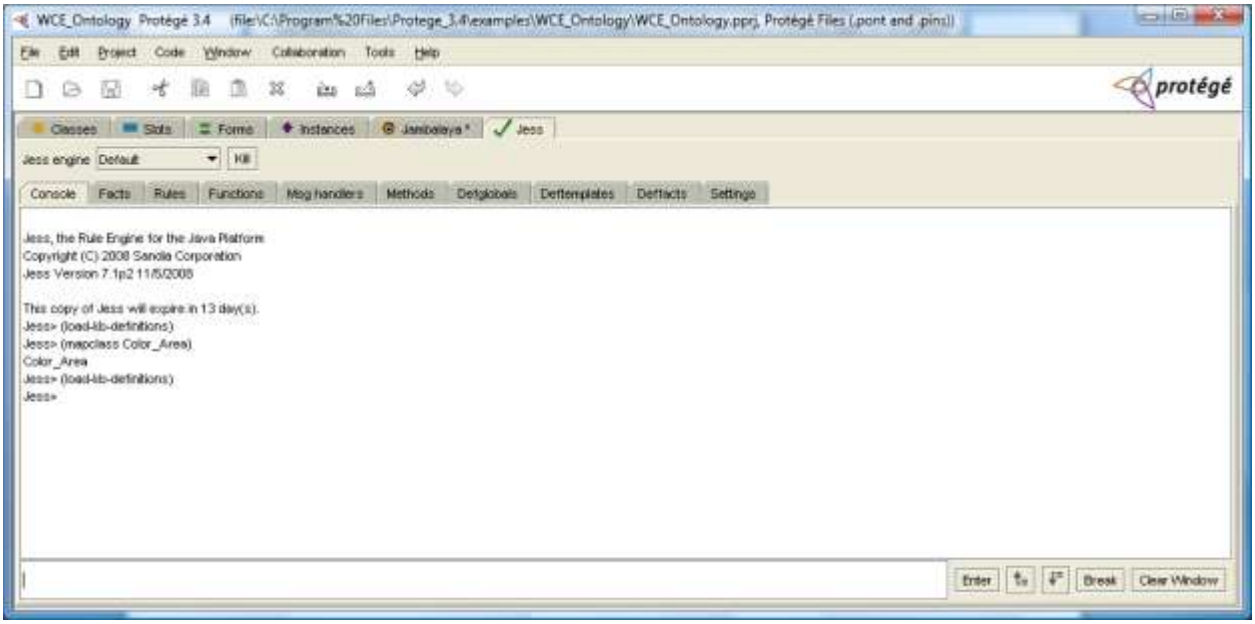
```
Jess> (run)
The patient may be suffering from bleeding 0.949999988079071because hue has higher value
The patient may be suffering from bleeding 0.8299999833106995because hue has higher value
The patient may be suffering from bleeding 0.8299999833106995because hue has higher value
The patient may be suffering from bleeding 0.949999988079071because hue has higher value
The patient may be suffering from bleeding 0.8999999761581421because hue has higher value
The patient may be suffering from bleeding 0.800000011920929because hue has higher value
6
Jess>
```



Facts:

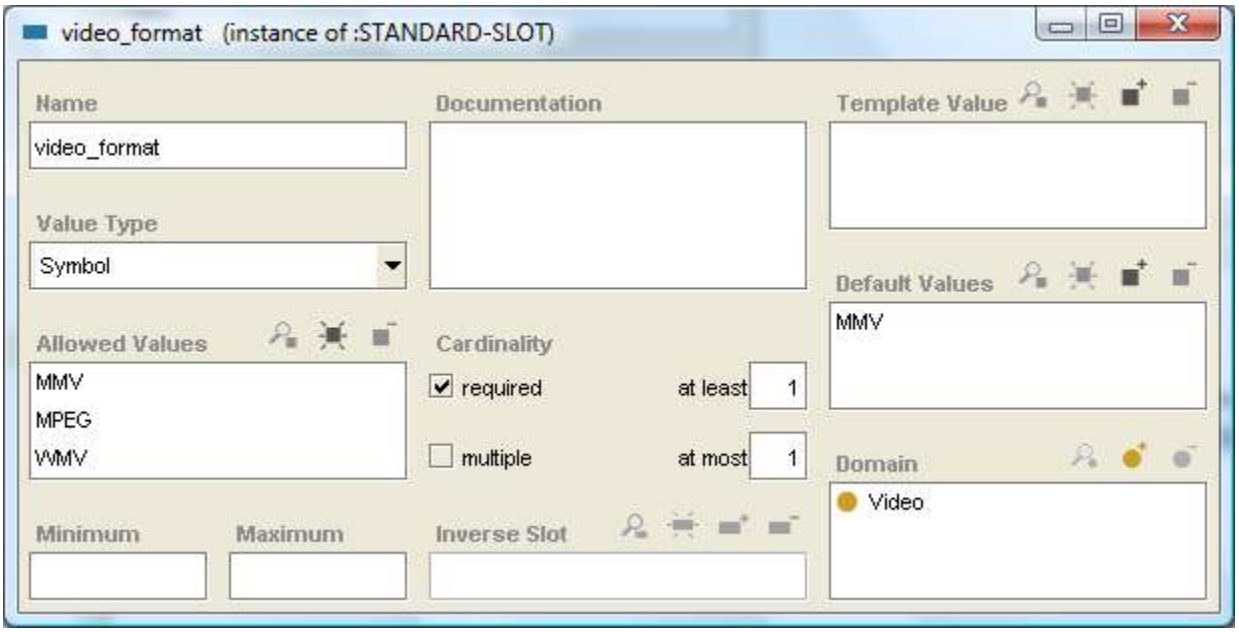


To load the jess rules: first it has to be mapped to class then type in jess console window load-kb-definitions; it will bring up the defined rules.



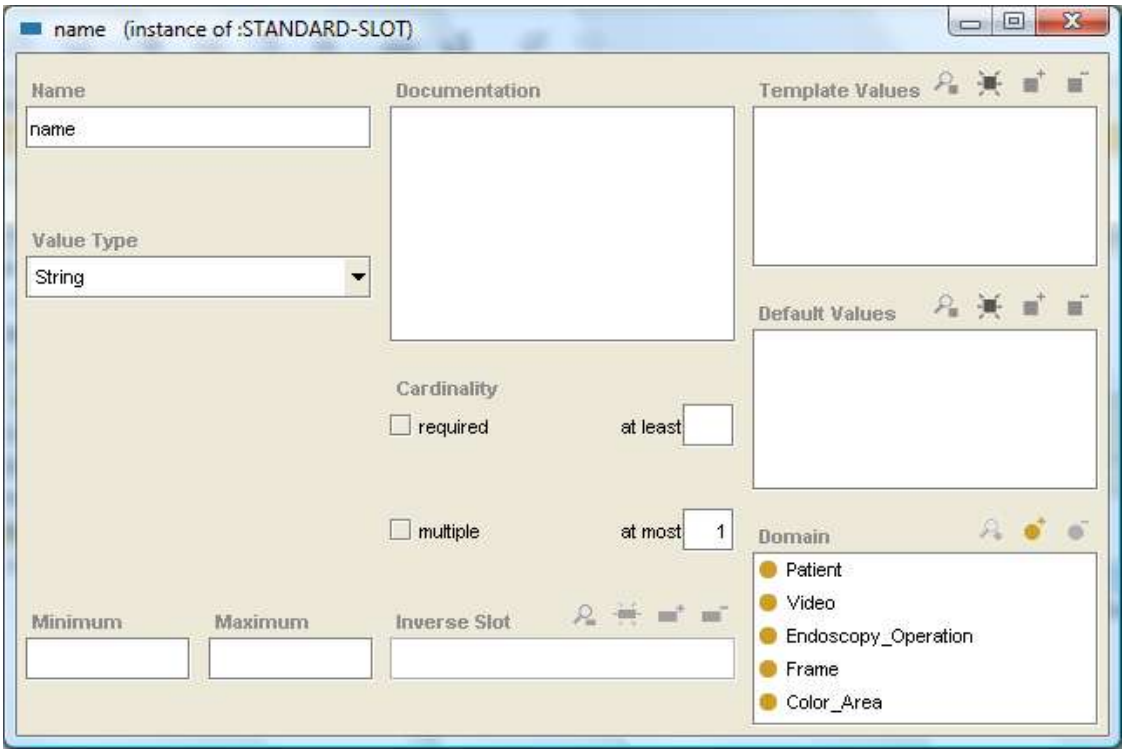
Assumptions:

By default MMV video format is selected:
T o do that go to video class→video_format property→ add the default value

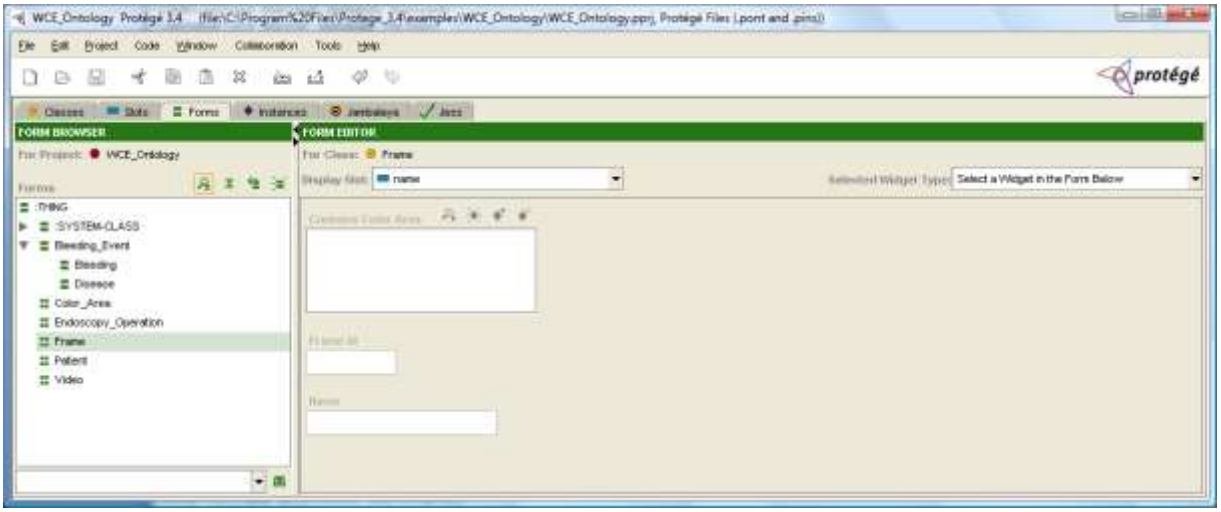


To change the instance name:

Add from patient class→name property→add the required classes in the domain so that name is displayed in all the classes



After that go to form and select name property (brighter one) from the display slot list box so that we can change the instance names do it for all classes:



Conclusion:

In this paper, I proposed a framework called multimedia ontology for wireless capsule endoscopy (WCE) videos. The proposed approach consists of three steps: (1) collection of raw data in WCE videos, such as video data formats, features, meta-data and anomalies, (2) classification of raw data into concepts including generic and specific ontology (3) identification of relationship between concepts such as 'Is-A', 'Part-Of', and 'Has-A'. The main contributions of the proposed approaches are summarized as follows:

- I build multimedia ontology for a real application using Protégé 3.4, i.e., wireless capsule endoscopy videos.
- Worked on Java Net Beans IDE 6.5 to call the classes and instances that has been created in protégé
- Worked on Microsoft Visio2007 to show the flow process of bleeding detection
- Worked on two different inference engines: Algernon, Jess and sample outputs is shown

References:

[1] Alejandro Jaimes, John R. Smith, "Semi-automatic, data-driven construction of multimedia ontologies", IEEE, Intl. Conf. On Multimedia and Expo, July 6-9, 2003, Baltimore, MD, USA.

[2] F. Vilarino , P. Spyridonos , O. Pujol , J. Vitria , P. Radeva "Automatic Detection of Intestinal Juices in Wireless Capsule Video Endoscopy", IEEE, 18th International Conference on Pattern Recognition (ICPR'06) Volume 4.

[3] Subodh K Shah, JungHwan Oh, Jeongkyu Lee, Xiaohui Yuan, Shou Jiang Tang, "Automatic classification of Digestive organs in Wireless Capsule Endoscopy videos", ACM SAC, 1041-1045, Seoul, Korea, 2007.

[4] Subodh K Shah, Pragya P Rajauria, Jeongkyu Lee, M. Emre Celebi "Classification of Bleeding Images in Wireless Capsule Endoscopy using HSI Color Domain and Region Segmentation", URI-NE ASEE 2007 Conference.

[5] M Mylonaki, A Fritscher-Ravens, P Swain, "Wireless capsule endoscopy: a comparison with push enteroscopy in patients with gastroscopy and colonoscopy negative gastrointestinal bleeding", Department of Gastroenterology, Royal London Hospital, Whitechapel, London, UK.

[6] G. Bresci, G. Parisi, M. Bertoni, T. Emanuele, and A. Capria. "Video capsule endoscopy for evaluating obscure gastrointestinal bleeding and suspected small-bowel pathology". J Gastroenterol, 39(8):803-806, August 2004.

[7] Henrik Eriksson IEEE Intelligent Systems "Using JessTab to Integrate Protégé and Jess", Linköping University

[8] Ernest Freidman-Hill. "Jess in Action: Java Rule-based Systems". Manning Press, 2003.